**OPERAȚII PE BIȚI**

Adunare: 1 + 1 -> 0 și carry 1

Scădere: 0 - 1 -> 1 și carry -1  
  
  
  
Înmulțire: Verific dacă se înmulțește cu o putere a lui 2 (2, 4, 8, 16).

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

bit - **bi**nary digi**t**baza x - are cifre de la 0 la x – 1

Numere pozitive:  
Numărul Y din baza X = , n este numărul de biți din reprezentare  
b0 e LSB, bn-1 e MSB  
  
Conversii:  
- Bold = 10 are 10 cifre (0 – 9) B -> Bp – Grupăm din B în câte P cifre, p =   
- Bnew = 100 are 100 de cifre (0 – 99)

Numere negative (Pentru care, uneori, e nevoie de circuite speciale)  
*|S|\_\_\_\_\_\_\_\_\_ - S e MSB, bit rezervat pentru semn, deci sunt disponibili n – 1 biți din reprezentare!*

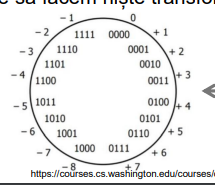
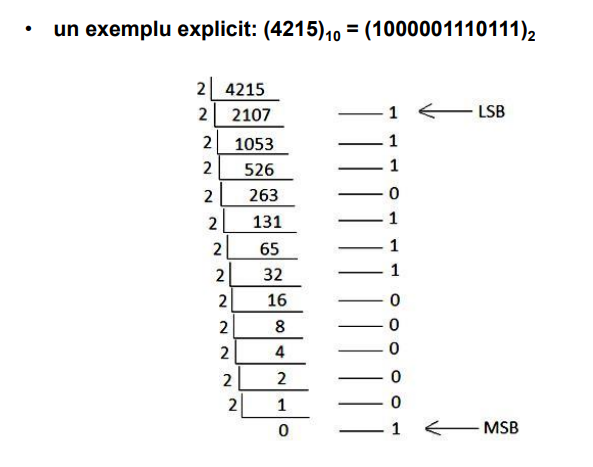
Numărul -Y din baza X =

Complement față de doi:

(-x)b10 -> b2 1. Scriem |x| în binar 2. Inversăm biții 3. Adăugăm 1

(-x)b2 -> b10 1. Scriem numărul așa cum e 2. Inversăm biții 3. Adăugăm 1

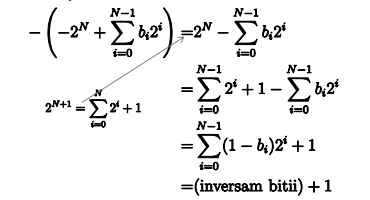
Folosim sistemul binar: Același algoritm de adunare, pot fi folosite aceleași circuite pentru adunare naturală

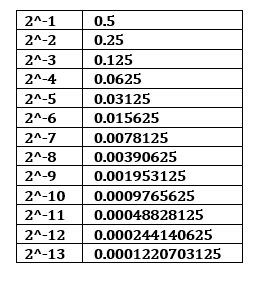
Cel mai mare număr care se poate reprezenta pe N biți: 2N – 1

Cel mai mare număr care se poate reprezenta (complement față de 2) pe N biți: 2N-1-1  
Cel mai mic: -2N-1  
MSB pe poziția: N – 1 (maximal în reprezentare)

X – Număr natural. Biți necesari pentru reprezentare: ceil(log2X)   
X – K cifre în HEX, de câți biți e nevoie pentru BIN: K \* log216 = 4 HEXold BINnew  
X – K cifre în BIN, de câți biți e nevoie pentru HEX: ceil(k / 4) BINold HEXnew  
X – K cifre în DEC, de câți biți e nevoie pentru BIN: ceil(k \* log210) DECold BINnew

***De ce funcționează complementul față de 2?***  


**BINARY FIXED-POINT**



**Ex: 101.101**

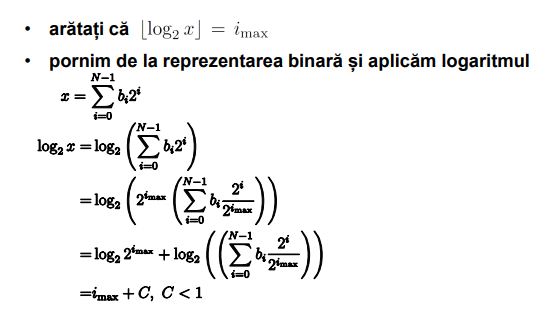
* 101.101 = 5.625

**Ex: 3.75**

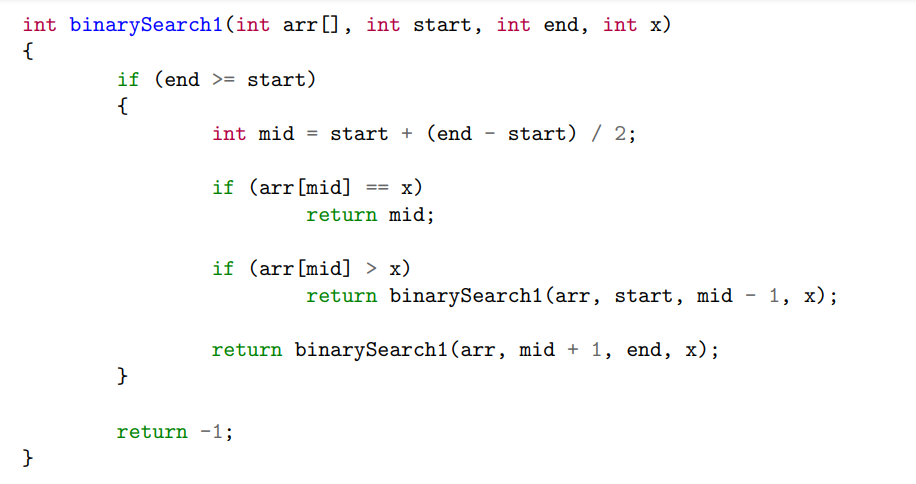
* 3.75 = 11.11

**LOGARITM ÎNTREG**

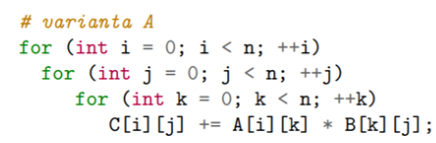




**OVERFLOW-SAFE BINARY-SEARCH**



**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(n3)  
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 \* n3

**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 arisotcrației | Limbajul Pascal e numit în onoarea lui

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

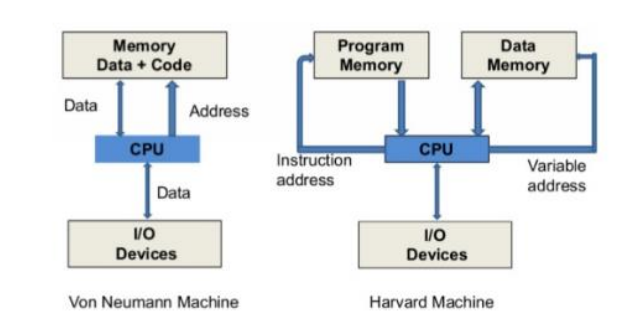
*George Boole* – Scrie ”The Laws of Thought” | Introduce logica booleană și analizează operațille logice de bază: Negație / Conjucție / Disjuncție / Disjuncție exclusivă | Ele stau la baza teoriei informației

*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: Principiul substituției Liskov + altele în distributed computing

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

*Diffie Hellman* – Schimbul de cheie Diffie-Hellman – Soluția în a trimite mesaje secrete într-un canal de comunicație 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 mașină care face un singur lucru la o mașină programabilă
* Design modular (pe module)
* Teorie despre ce este posibil pe aceste mașini
* Dorința de a face lucrurile optim, la limită și fără risipă

După Al Doilea Război Mondial, dorința de cercetare în domeniul calculatoarelor crește în ritm exponențial

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

**TEORIA PROBABILITĂȚILOR**

P= (atunci când evenimentul nu este influențat de mediul exterior)

(probabilitatea ca două evenimente **INDEPENDENTE** să se întâmple, 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.

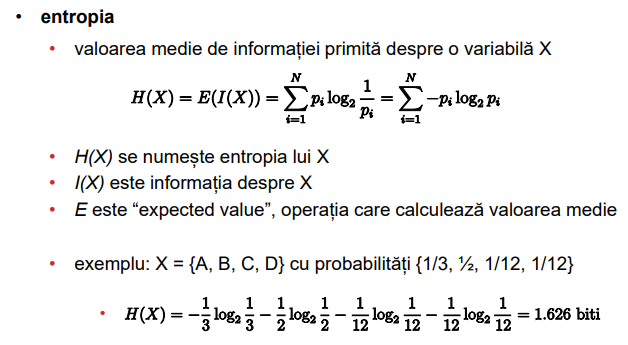
**TEORIA INFORMAȚIEI**

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

(Câtă informație ne oferă o valoare în funcție de probabilitatea ca ea să apară)

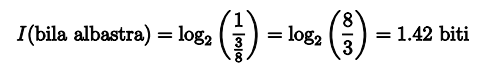
Calculăm cantitatea de informație folosind X (variabila aleatoare):  
N – Numărul de valori distincte: x1, x2 ... xn  
P - Fiecare valoare apare cu probabilitatea p1, p2 ... pn   
 pi mai mic -> Obținem cantitate mai mare de informație

Putem privi formula astfel: = rezultat în biți!  
N – Numărul total de evenimente  
M – Numărul de evenimente favorabile



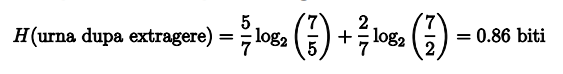
I) 12 piese: 3 cuburi, 3 conuri, 3 sfere, 3 piramide | Fiecare formă are una dintre culorile: roșu, galben, albastru

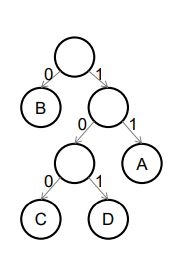
A extrage o piesă roșie. Câtă informație primește copilul B, care vrea să afle tipul piesei?

Ințial: 12 variante, dar 4 piese sunt de culoare roșie, deci pi =

=(3) (xi este o piesă roșie)

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

Se extrage o bilă albastră.  
Primim I(bilă albastră) cantitate de informație

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:  
Frunzele – Codurile  
Stânga/Dreapta – Decis de 0/1  
  
Implementare: Algoritmul lui Huffman   
*(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 codare mai scurtă pentru event. 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: log2(a!) a log2(a) – a log2e

**Probabilitățile pentru fiecare caz:**(În principiu : Cât știm / Cât vrem să știm)

a) x are exact două valori "1" în reprezentarea sa binară: , pp. n par.

b) x are exact N/2 valori "1" în reprezentarea sa binară (N par): , pp. n par.

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

, pp. 4 | n

d) x are MSB (Most Significant Bit) setat la "1":   
e) x este impar:   
f) x este o putere a lui 2:   
g) x are primii N/2 biți din reprezentarea sa binară setați la "0" (N par):

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

i) x are un număr par de biți setați la "1" (N par):   
j) x = 42:

**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 \* 232 + %eax)

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

jmp \*%eax – Sare la o adresă

aritate 1 – Operația se aplică unui singur număr

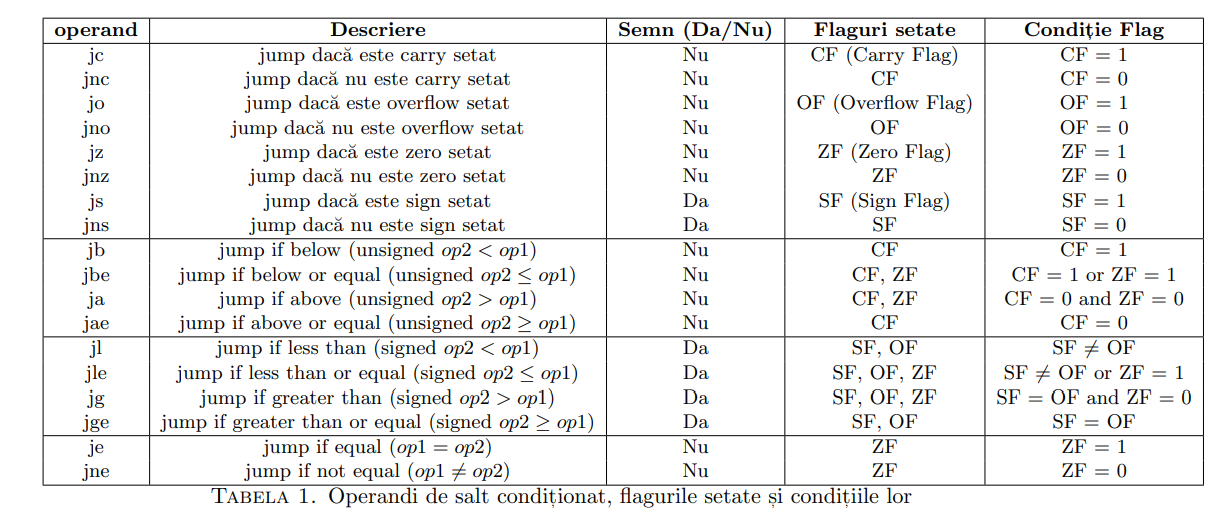
.asciz -> Lungimea șirului + 1 pentru \n

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

**¬** NOT (are aritate 1) **∧** AND **∨** 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

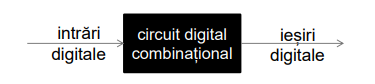
a(b, c, d) = a + b + c · d



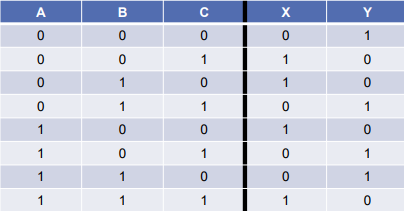
**CIRCUITE**

**CIRCUIT DIGITAL COMBINAȚIONAL**

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

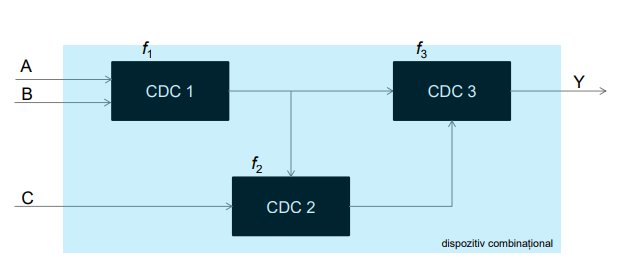
Eficiente în general. Problema majoră: sunt one-shot  
- Nu putem itera  
- Nu permit niciun fel de logică internă / memorie internă (nu are stări interne)  
- Sunt prea simple: Pui un semnal digital constant la intrare și ai un semnal digital constant la ieșire  
- Logica combinațională este insuficientă pentru anumite implementări

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

Pentru fiecare intrare, trebuie să știm care e ieșirea  


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 = f3(f1(A, B), f2(f1(A, B), C)) – Ne uităm ce intră în Y șamd.

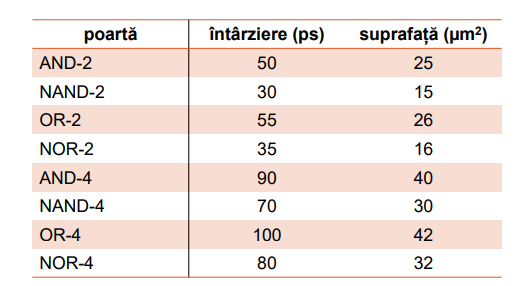
Timpul total de propagare: tp, total = tp, 1 + tp, 2 +tp, 3 (cea mai lungă cale)  
*Timpul maxim după care avem o ieșire validă dacă avem intrări valide*

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 porți logice**

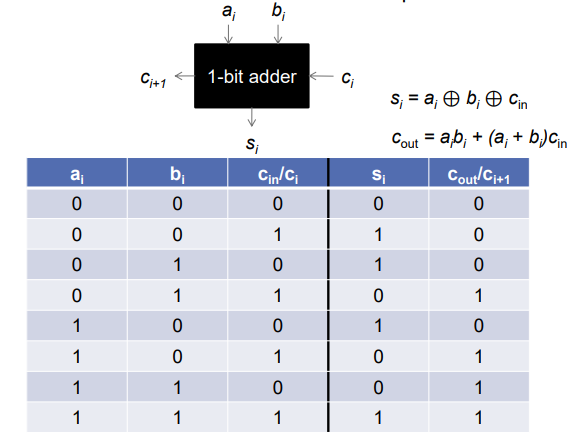
|  |  |  |
| --- | --- | --- |
| Operația | Valori | Notație |
| NOT |  | (A bar, A complement) sau !A |
| AND |  | AB |
| OR |  | A + B |
| XOR |  | A ⊕ B |
|  | | |

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

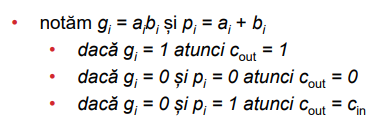
Tranzitoare CMOS: Circuitele analogice sunt mai eficiente pe logică negată (mai puține componente electrice)

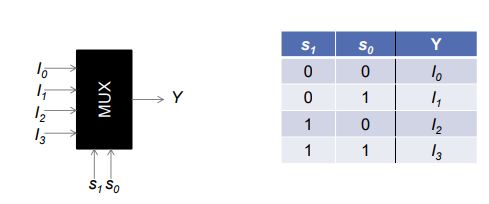
**CIRCUIT INPUT OUTPUT NR INTRĂRI NR IEȘIRI   
ADUNARE A (N biți) B (N biți) S (N + 1 biți) 2N N + 1**

Cât de mare va fi circuitul? (N + 1)

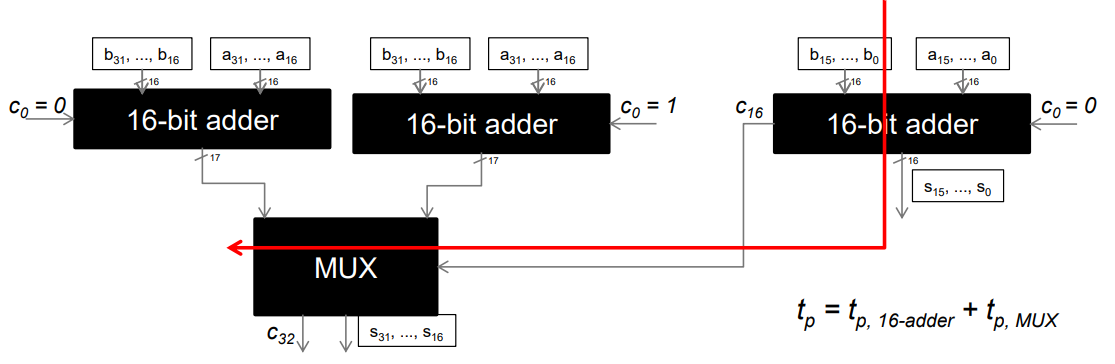
Putem defini un circuit bloc fundamental pe care bazăm totul:

Folosim circuitul în cascadă  
tp =

  
Trebuie așteptată calcularea biților de carry  
  
Putem generaliza

**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: 2n -> n linii de selecție

Îmbunătățire: Adunare pe 32 de biți

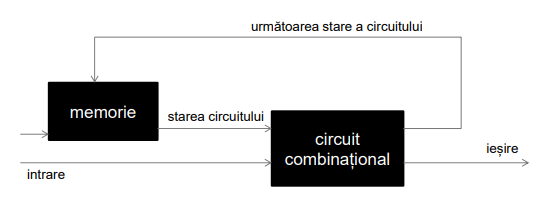


Putem aplica aceeași idee pentru circuitele de 16 biți de mai sus: tp =

**CIRCUITE SECVENȚIALE**

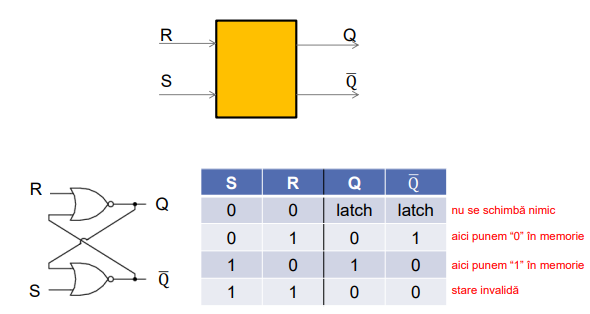
Permit elemente de tip ”memorie” > putem adăuga o stare circuitului (există stare internă)

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



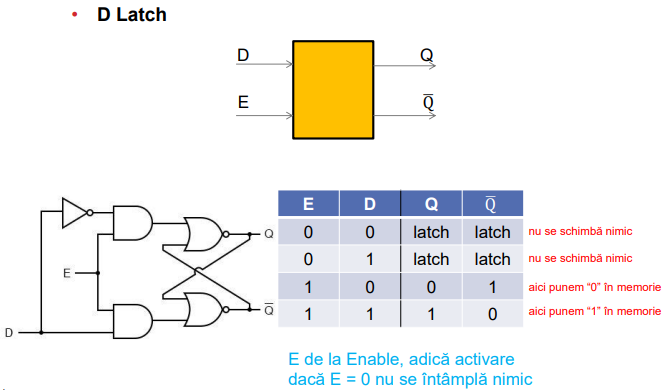
Biții pe care îi reprezentăm - voltaj.  
Energia electrică - dificil de stocat (provoacă fenomenul de scurgere / leakage)  
Pentru a memora ceva > Refresh din când în când pentru actualizarea nivelului de energie electrică

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



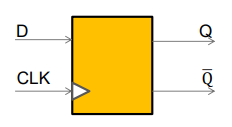
**D Latch** | Bun, are o intrare dar vrem să sincronizăm mai multe dispozitive

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

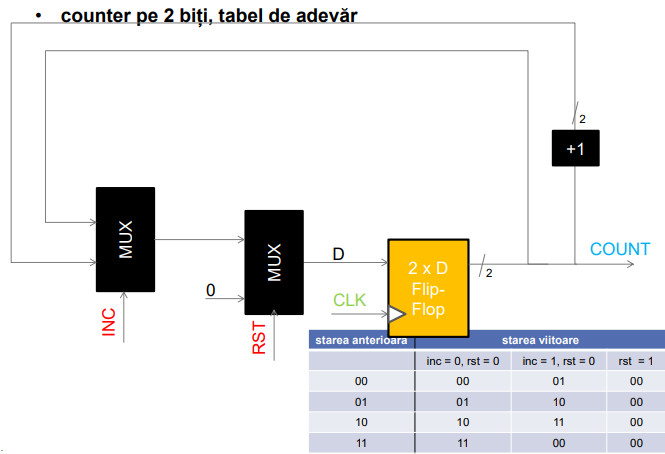


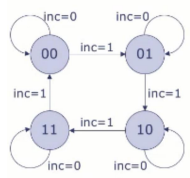
**D Flip-Flop**

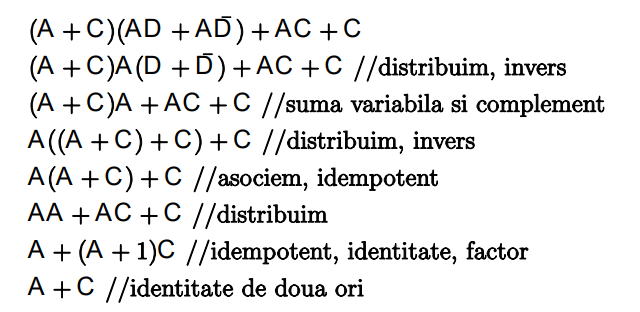
**E** devine clock (ceasul sistemului)   
CLOCK: La un interval fix de timp (la un ciclu), sistemul face ceva

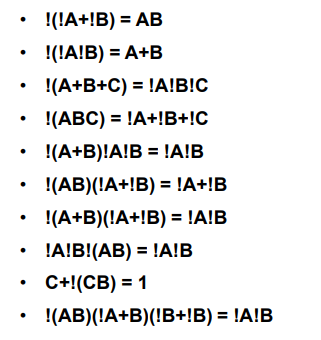


**Registru** = Un set de câteva D Flip Flops care au același CLK

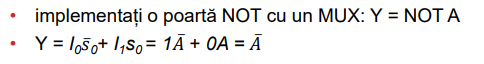
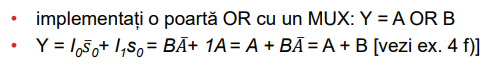


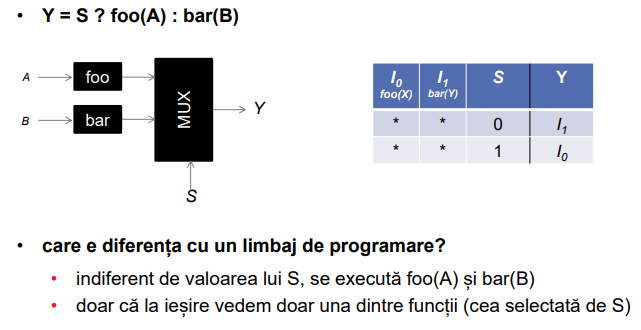
Stările

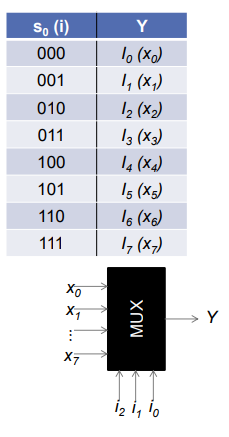
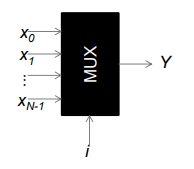
**LEGILE DE MORGAN SIMPLIFICĂRI**







  
Vrem să accesăm x[i]  
 Intrări: Vectorul x  
Semnal: Indexul i  
Dimensiunea intrării: N – 1 – 0 + 1 = N  
Dimensiunea lui S: ceil(log2N)

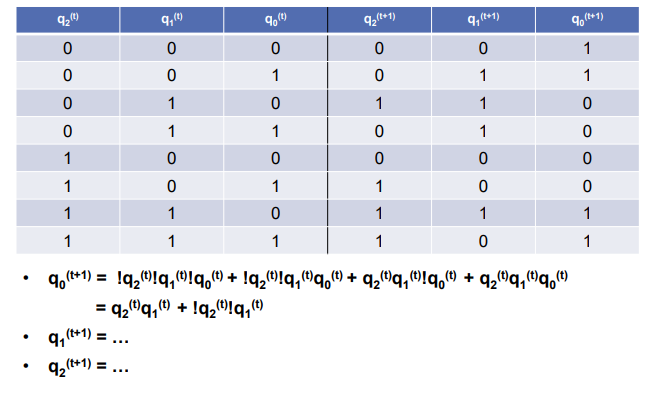
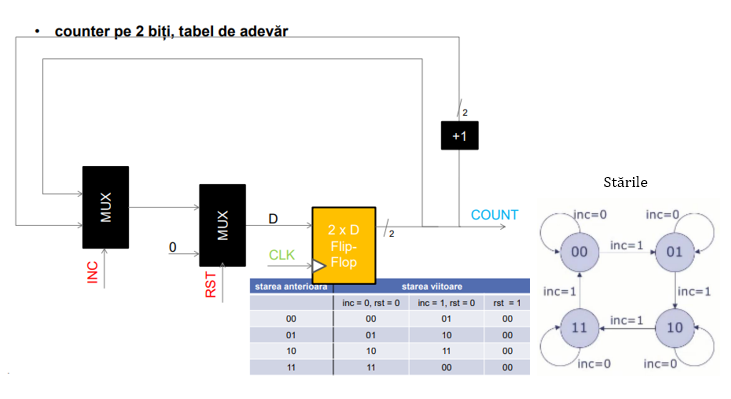
Câte MUX cu 2 intrări pentru a simula MUX cu N intrări?  
N – 1

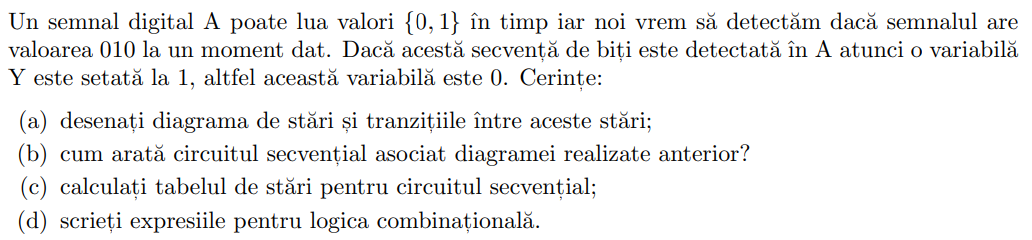
**Circuit secvențial COUNTER pe 2 biți**  
  
Relația dintre starea anterioară și starea viitoare  

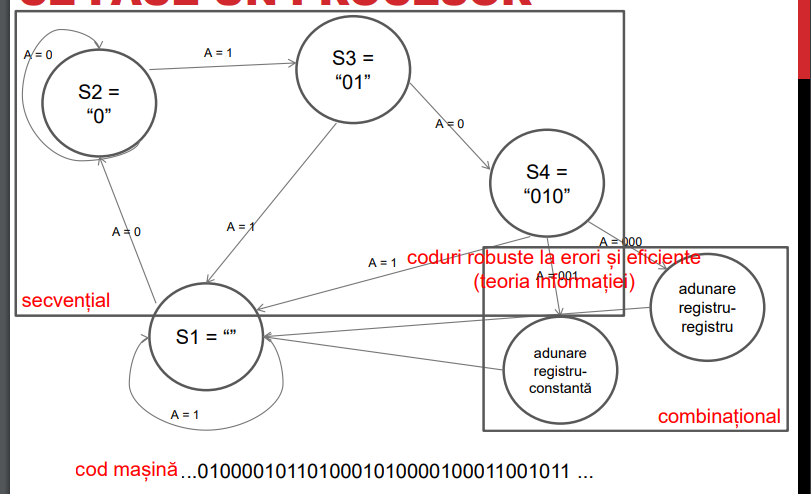

Desenul circuitului secvențial (stările sunt q0 și q1)



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).

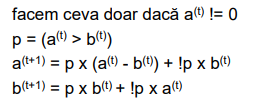


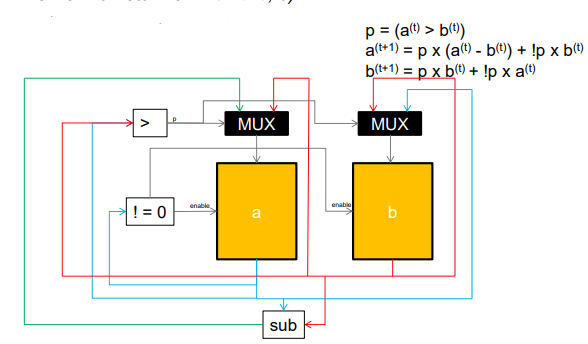




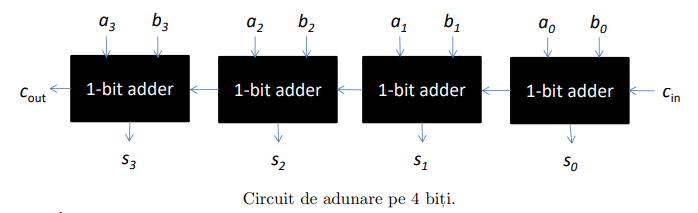
**Circuit secvențial pentru CMMDC**

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

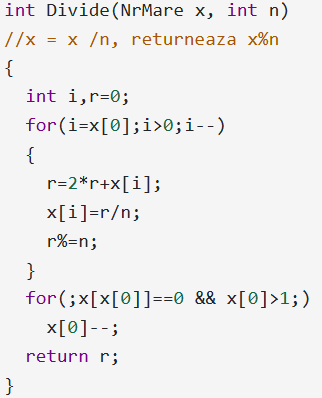
Avem variabilele: a(t), b(t)  
Ecuațiile de evoluție:  


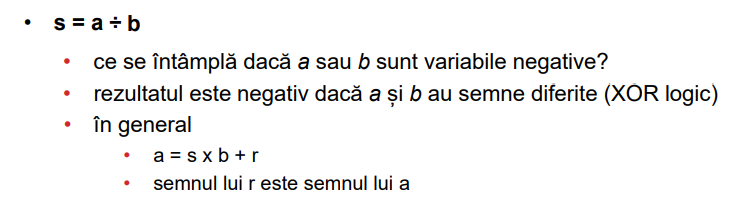


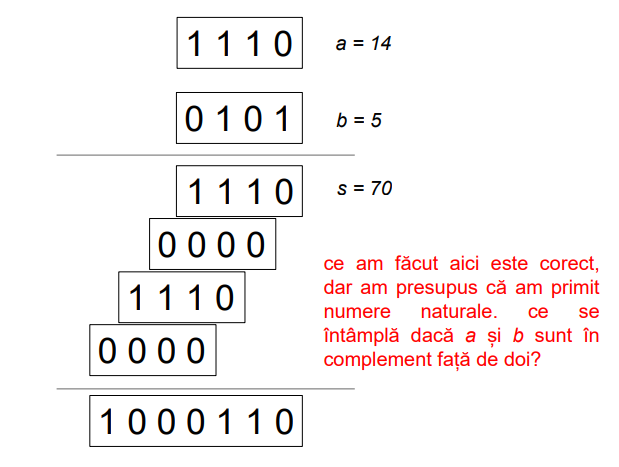
**Circuit de adunare pe 4 biți**



**ÎMPĂRȚIREA NUMERELOR ÎNTREGI**

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


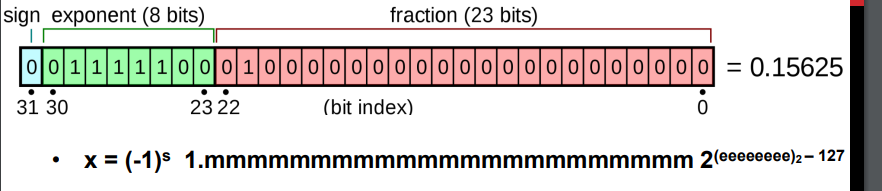


**ÎNMULȚIREA NUMERELOR ÎNTREGI**  
De la dreapta la stânga  
Dacă suntem în B pe bit 1, copiem A  
Dacă suntem în B pe bit 0, punem 0 peste tot  
  
**A și B naturale**  


**A si B întregi**  
Extindem numerele:

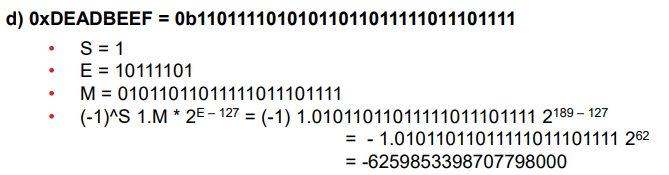
A = -2 A = 1110 A = 1111.1110  
B = 5 B = 0101 B = 0000.0101  
Se calculează la fel ca la numere naturale, dar **rezultatul este în complement față de doi.**

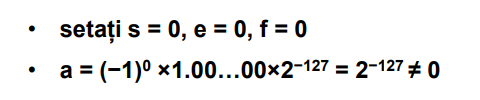
**FLOATING POINT**

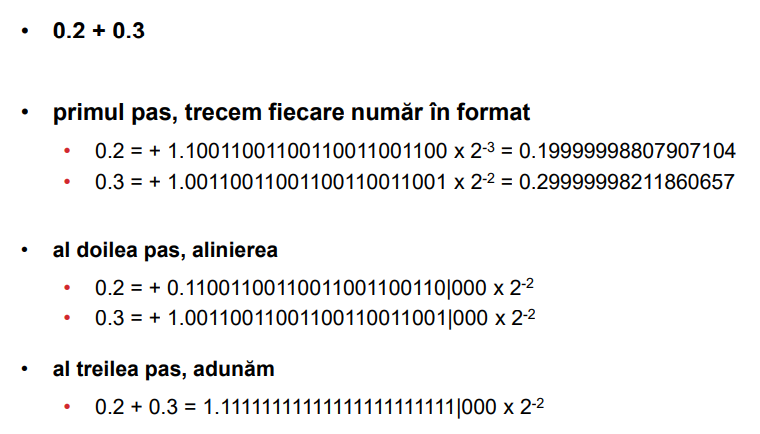


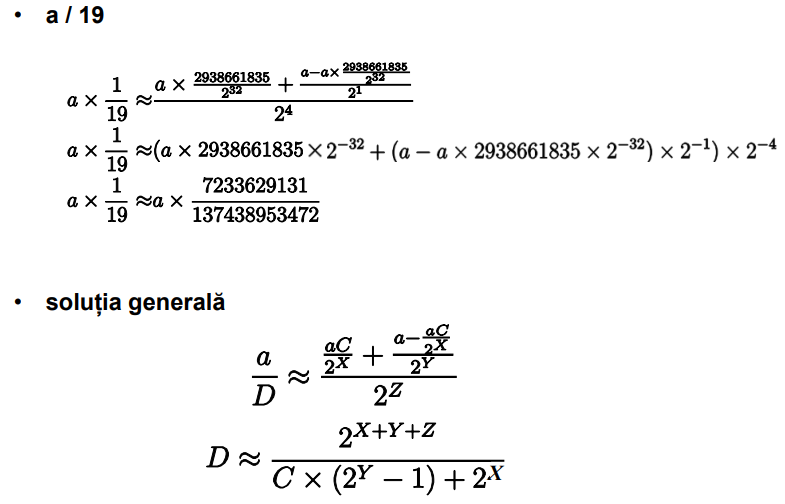
Schimbați semnul lui a: a = a ^ (1 << 31)  
Exponentul pe: 0x7F80.0000  
Extragem exponentul: (a & 0x7F80.0000) >> 23  
***Pentru a împărți la 4  
Exponent > 1 => Exponent = Exponent – 2 | Altfel: a = 0  
a = (a & ~MASK) | (exponent << 23)***

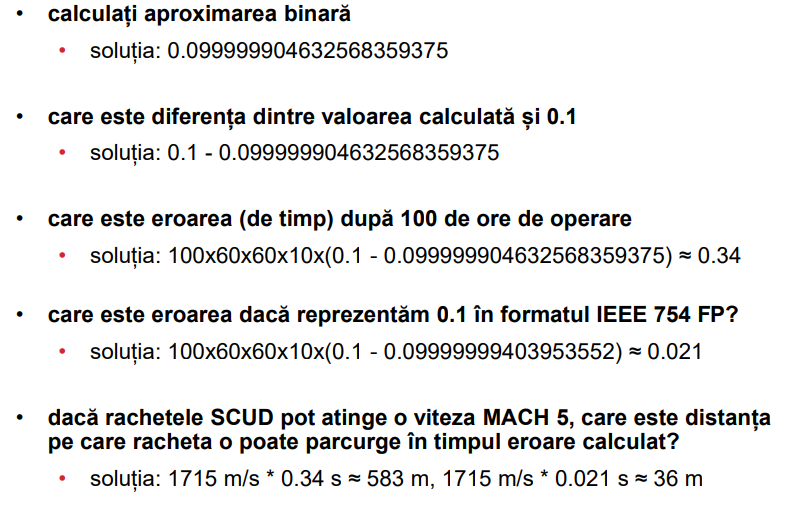
|  |  |
| --- | --- |
| SEMN | a >> 31 |
| EXPONENT | (a >> 23) & 0x000000FF |
| MANTISĂ | a & 0x007FFFFF |
| Nr. în baza 10 cu mantisă de M biți | ⌊log10 2 × M⌋ ≈ M/3 |
| abs(a) | a = a & ~(1 << 31) |
| a x 2 | a << 1 sau a + a |
| a x 2 | (a < 0) ? -((-a) << 1)): a << 1 |
| a x 16 | a << 4 |
| a x 3 | a << 1 + a |
| a x 7 | a << 3 - a |
| a / 8 | a >> 3 |
| a % 16 | a & 0x000F |
| a % m | a&(m − 1) |
| a / 16 | (a & FFF0) >> 4 |
| a x 72 | a << 6 + a << 3 |
| a / 16 + a % 16 | a & FFF0 + a & 000F |

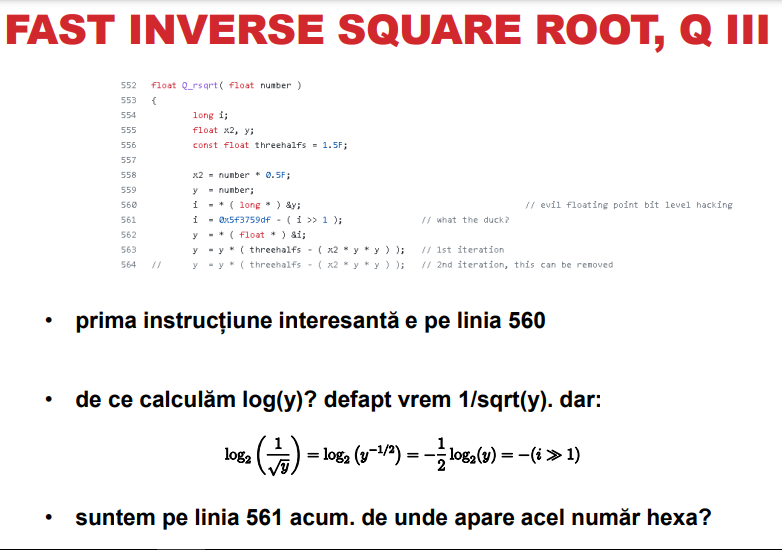


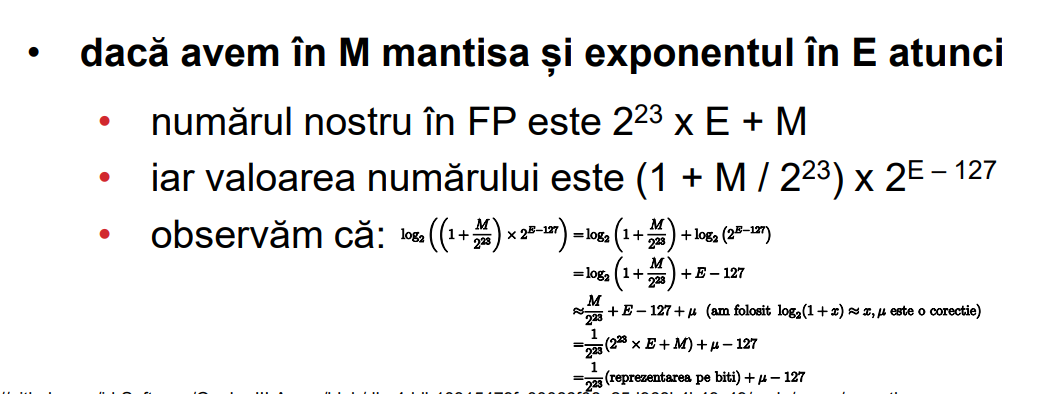


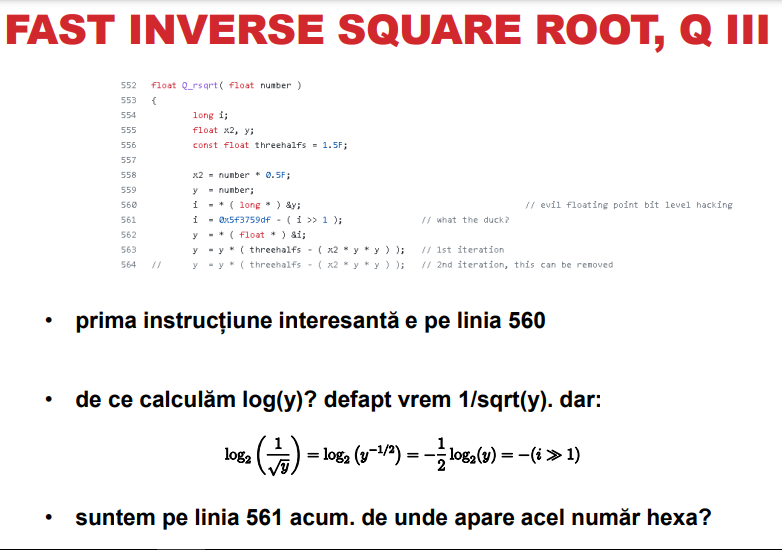


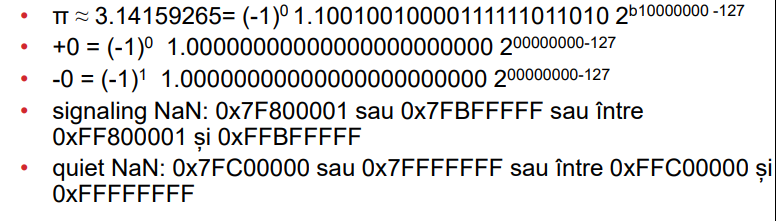




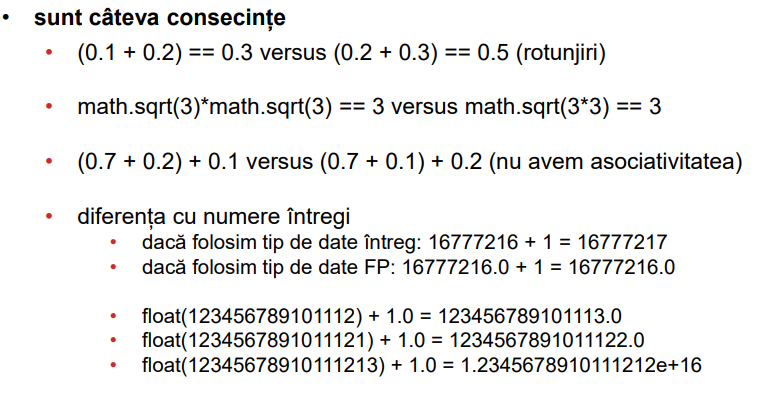








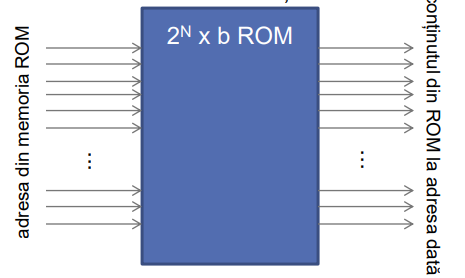
**CONSECINȚE FLOATING POINT**



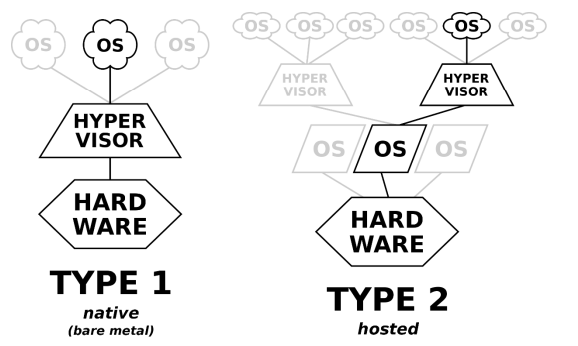
**ARHITECTURA CALCULATOARELOR MODERNE**

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 placa 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

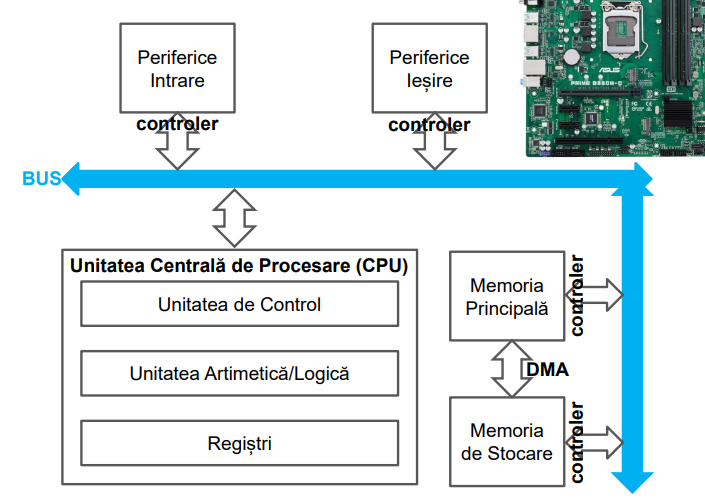
Componente  
TOT BIOS: Scris în ROM, câteodată în Programmable ROM, Erasble Programmable ROM  
„Să scriem în ROM”: „burning” sau „flashing” the ROM  
Este un circuit combinațional

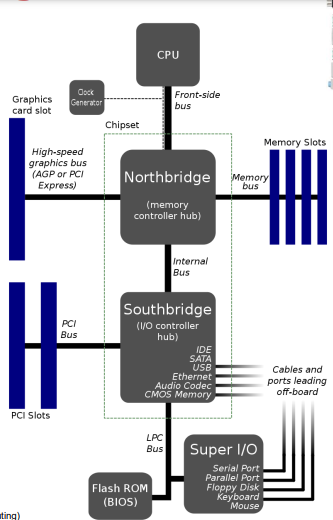
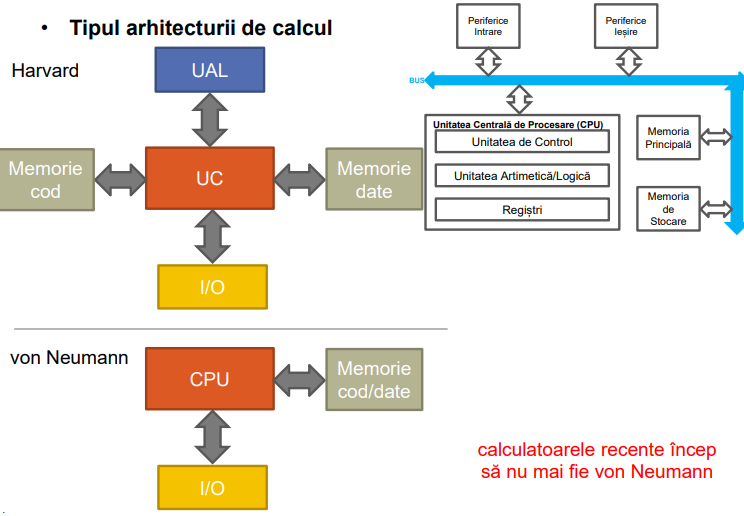


OS preia controlul de la BIOS  
 - Doar OS are acces la periferice, prin drivere  
 - Virtualizare/Emulare/Containere(Dockere)

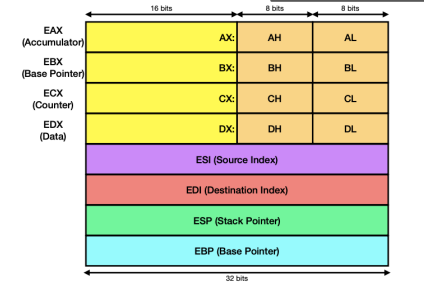


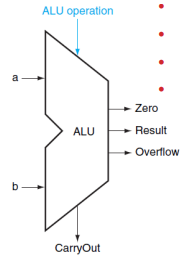
OS: Oferă imagine abstractizată a memoriei pentru fiecare proces pornit.  
OS pornește > Sistemul de calcul intră în ciclul obișnuit de procesare   
(Secvența boot se încheie)

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  
  
  


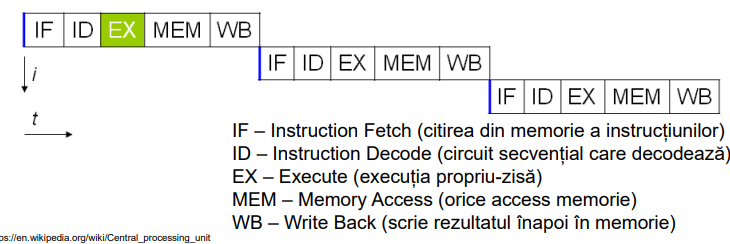


**I. CPU (5 componente)** – Creierul unității de calcul / Execută instrucțiuni  
 1. Clock   
 - Circuit special care generează „ceasul”  
 - Frecvența la care operează CPU (calcule + sincronizarea componentelor secvențiale)  
 - Frecvență mare -> Mai bine | Se măsoară în MHz sau GHz

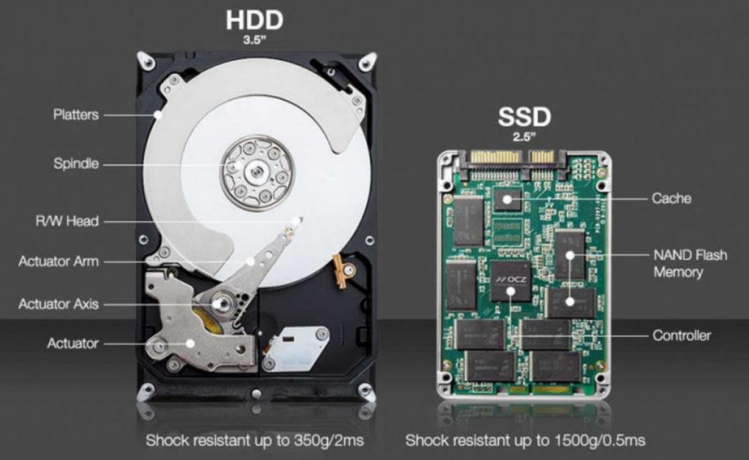
2. Regiștri („memoria”)  
 

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



**II. Memoria principală**- Conține cod și date  
- Este volatilă  
Tipuri - Static RAM (SRAM)  
 - Bazat pe flip-flops, rapid, scump, regiștrii din CPU de același tip  
 - Dynamic RAM (DRAM)  
 - Fiecare bit reprezentat prin combinație de tranzitor + condensator (condensatoarele suferă de leakage – scurgeri 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 (timp de refresh / în cât timp de la citire sunt disponibilizate datele)  
 - Consumul de energie

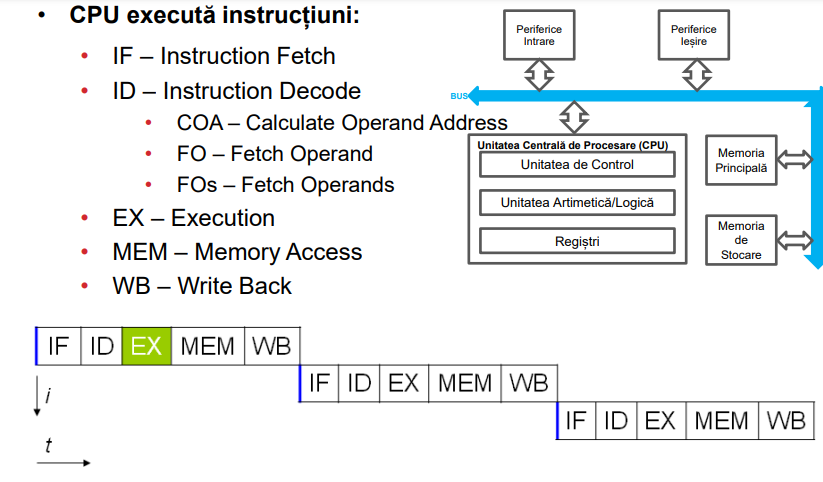


**III. Memorie de Stocare**

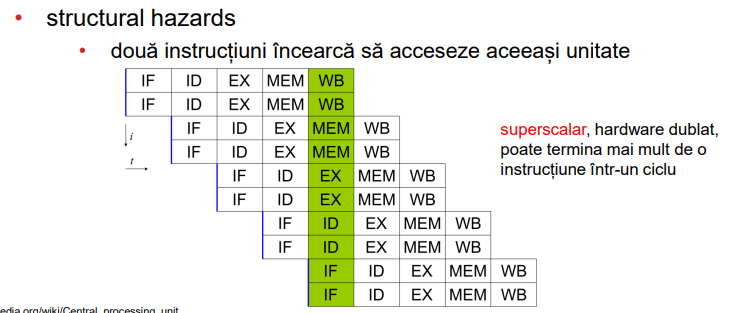
**-** Conține cod și date, este nevolatilă  
- SSD (Solid State Disk): Memorie flash, rapidă, scumpă, scriere mult mai lentă decât citirea  
HDD (Hard Disks): Mecanic

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

**PIPELINING**

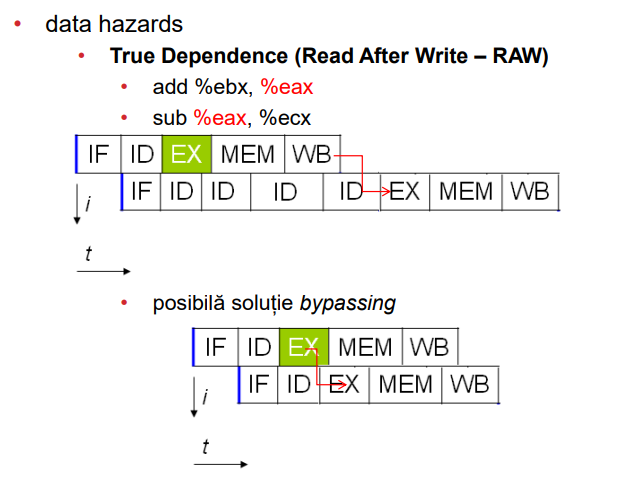


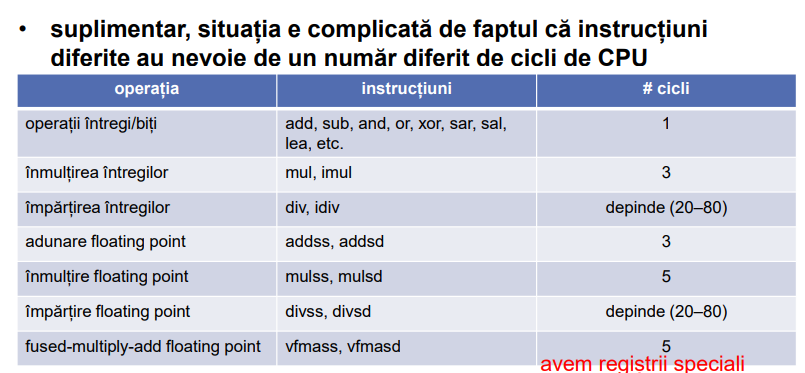
Pipeline stalls = Întârzieri în conducta de date  
Aceste evenimente se numesc hazards (erori)

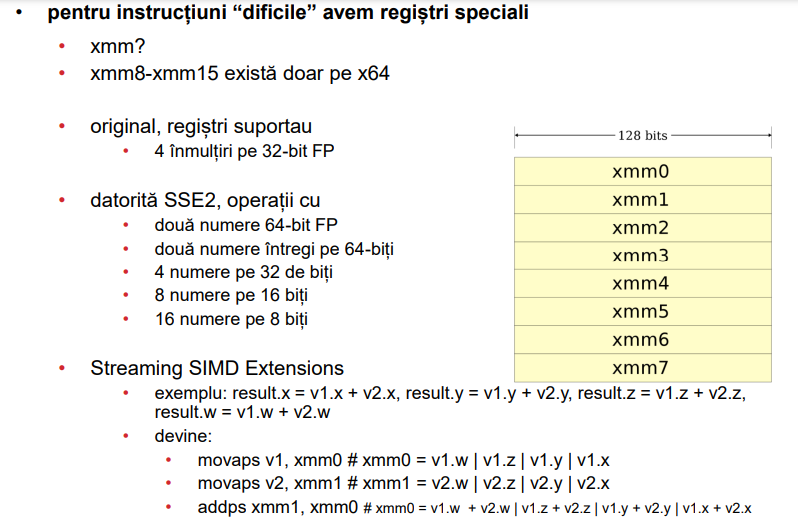
Structural Hazards – O unitate de calcul este deja utilizată  
  
I. Structural Hazards  


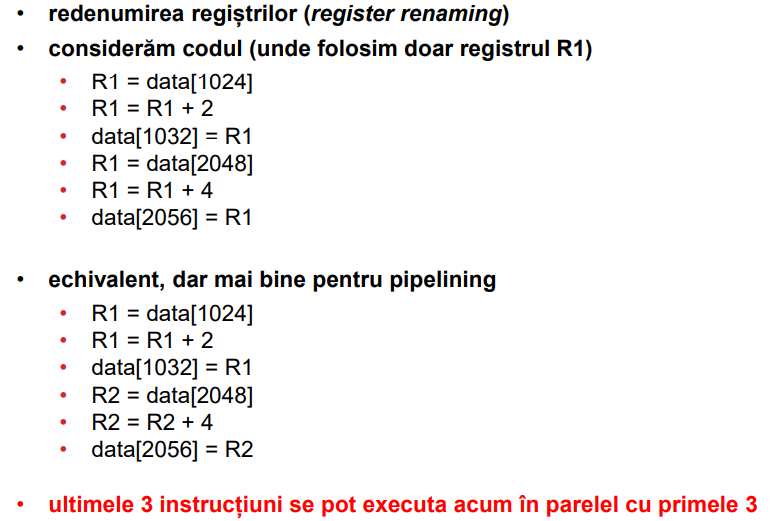
2. Data Hazards – Datele nu sunt pregătite pentru utilizare

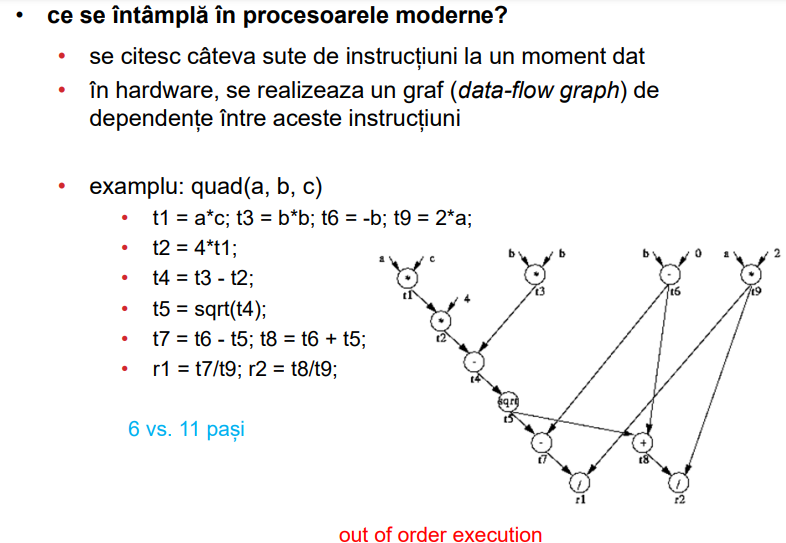






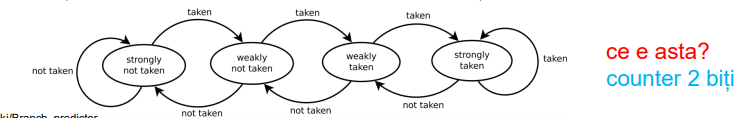






BRANCH PREDICTION





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

--

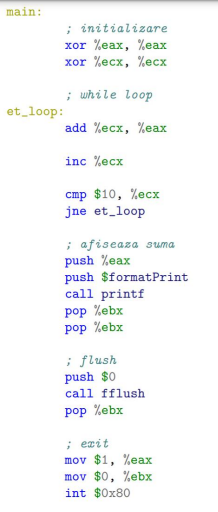
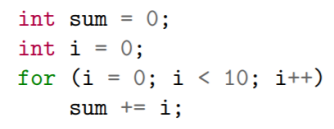
Adresa de memorie cea mai accesibilă 232 = 4GB (4.294.967.296 bytes)

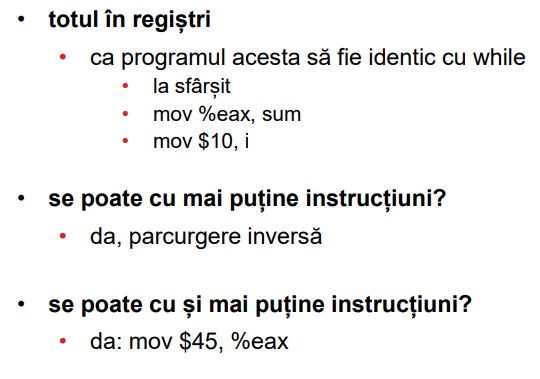
Adresa de memorie cea mai accesibilă pentru ***jne etichetă cu OPCODE 0110*** 228

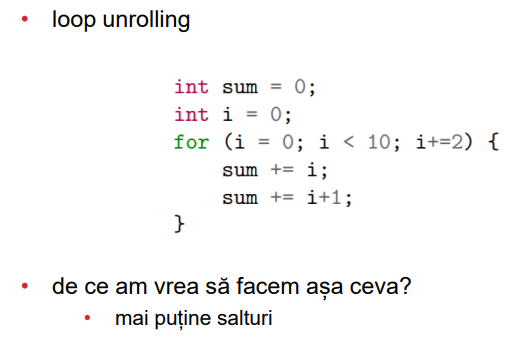
Regiștri suportați de add R1 R2 (OPCODE: 0011) 214

Regiștri suportați de add R1 R2 R3 (OPCODE: 0100) 9.33 biți pentru fiecare reprezentare a unui registru: Două poziții suportă 29 | O poziție suportă 210

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

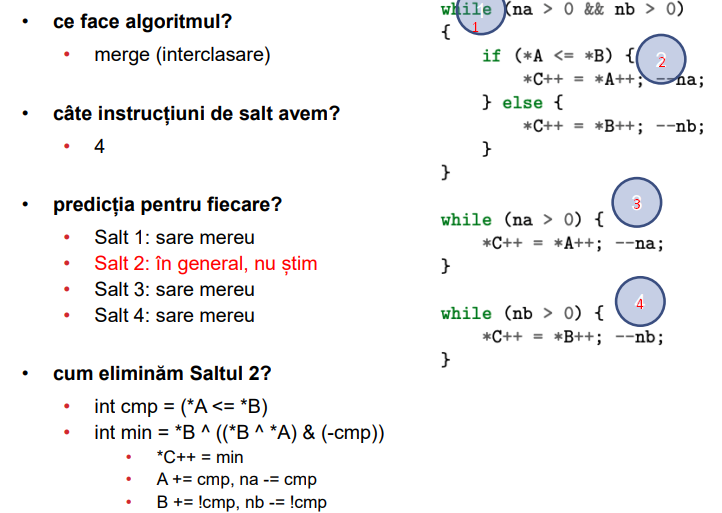
   
»

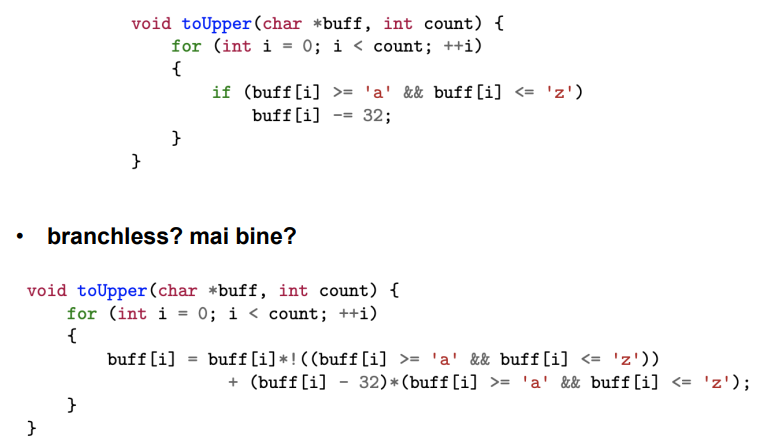
»

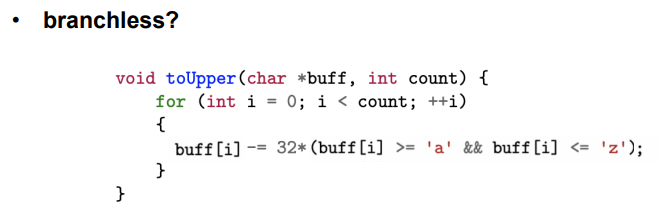
**» LOOP UNROLLING**



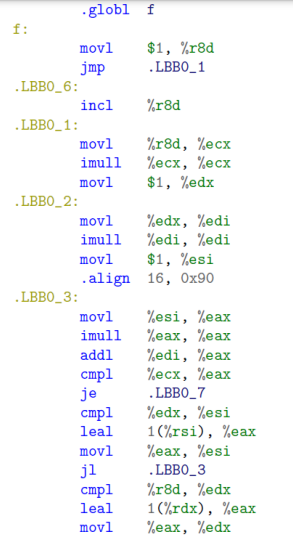
**ALGORITM DE INTERCLASARE / MERGE**

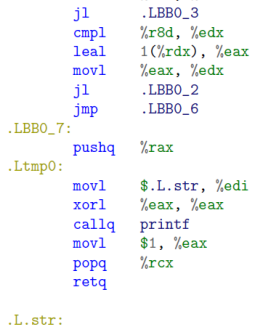


**TOUPPER()**



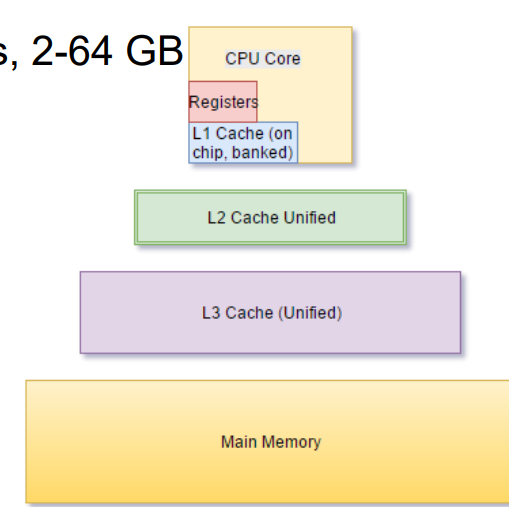
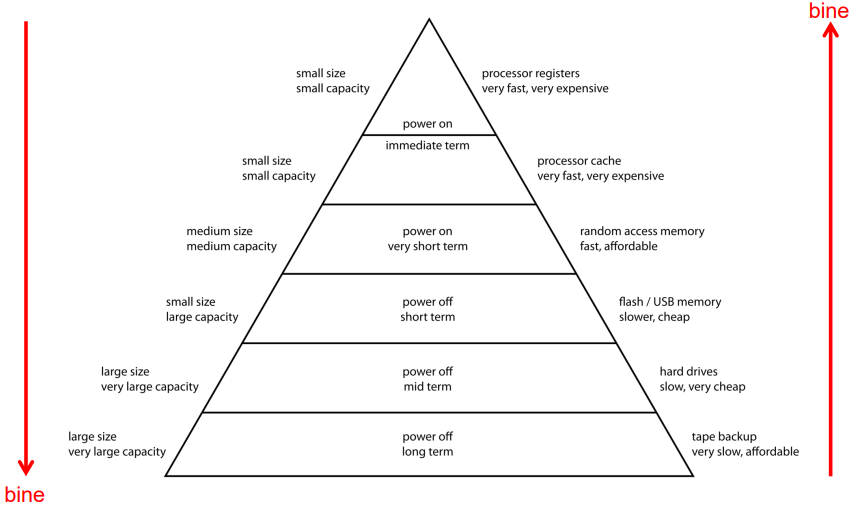
**VERIFICARE X2 + Y2 = Z2 PENTRU X <= Y**

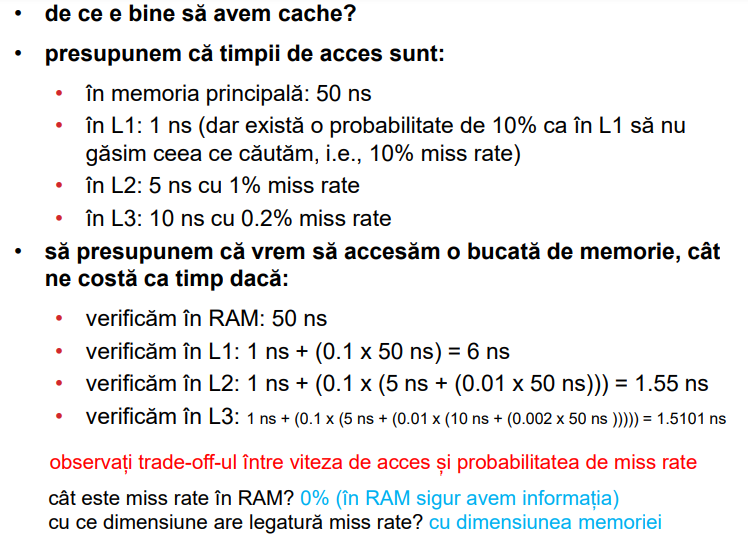


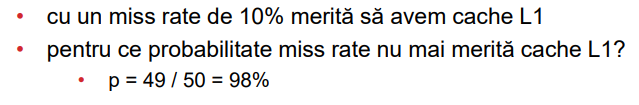


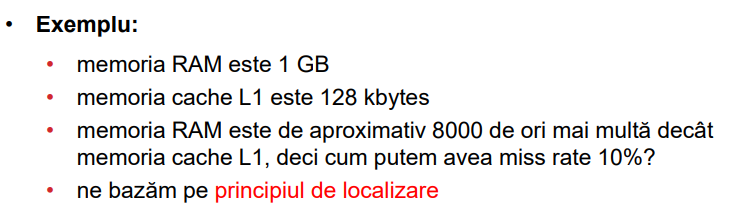
**IERARHIA MEMORIEI**

|  |  |  |  |
| --- | --- | --- | --- |
| **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 |

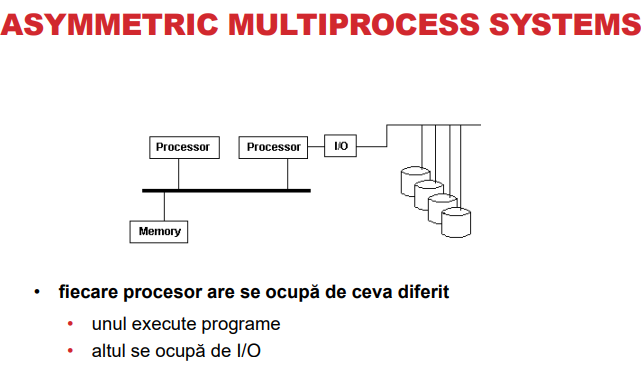
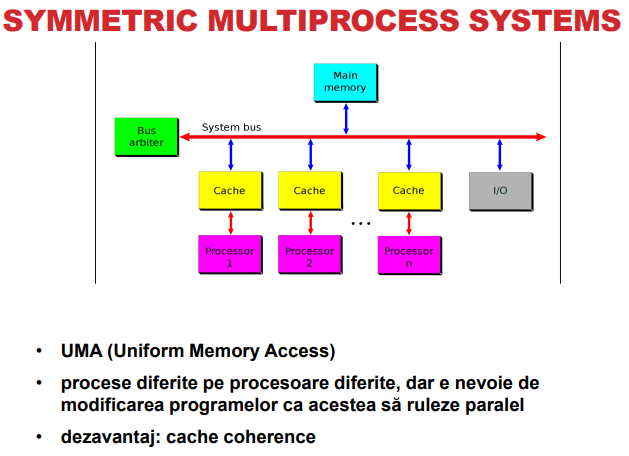
 

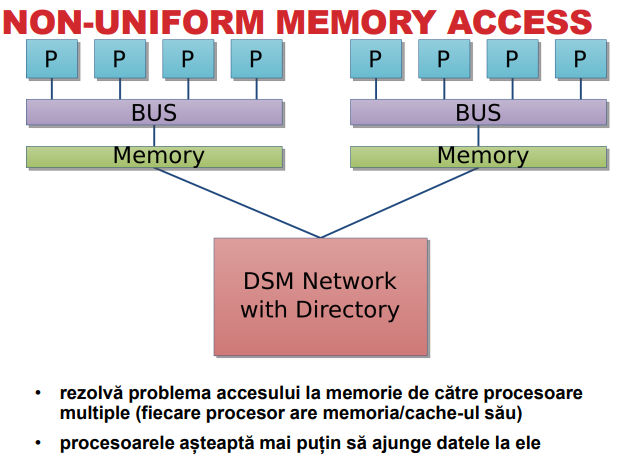


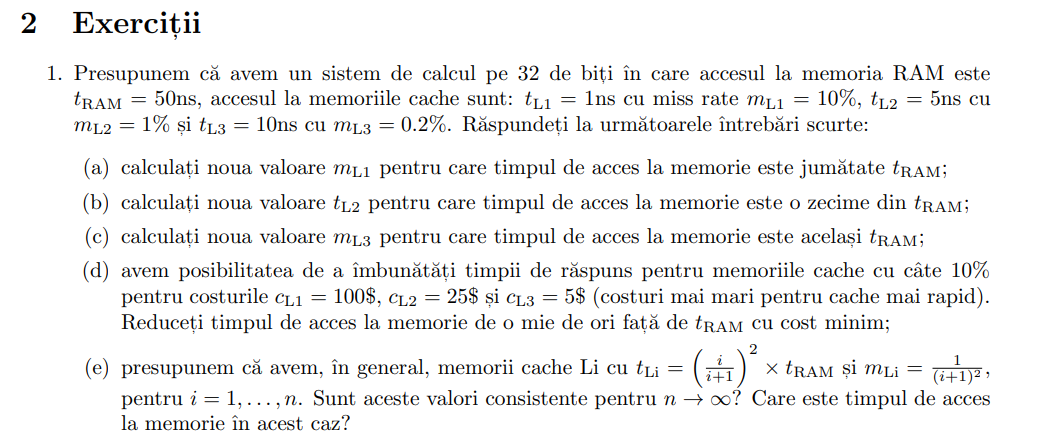


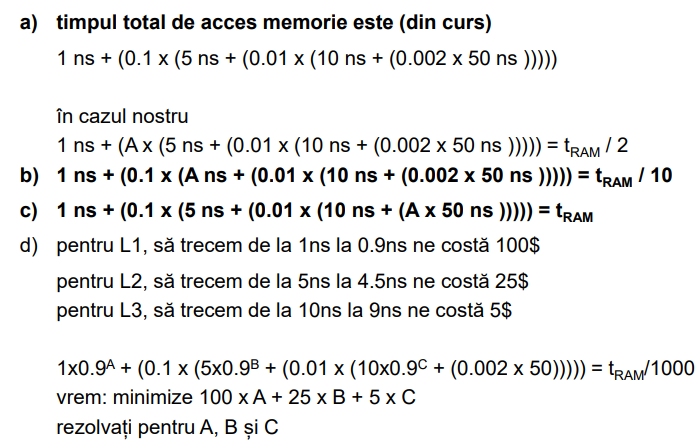






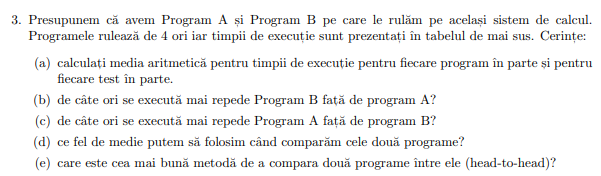


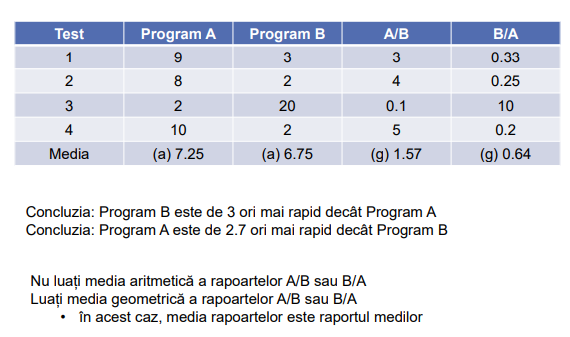
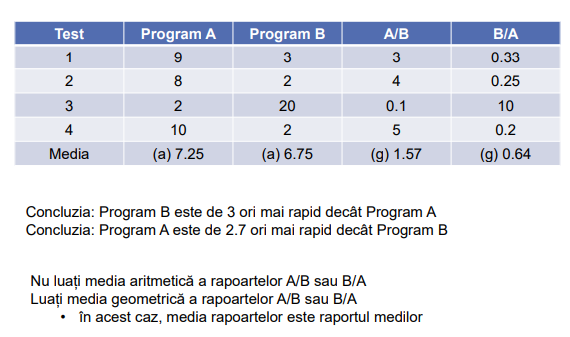


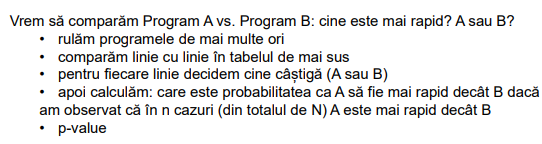
Criterii de performanță

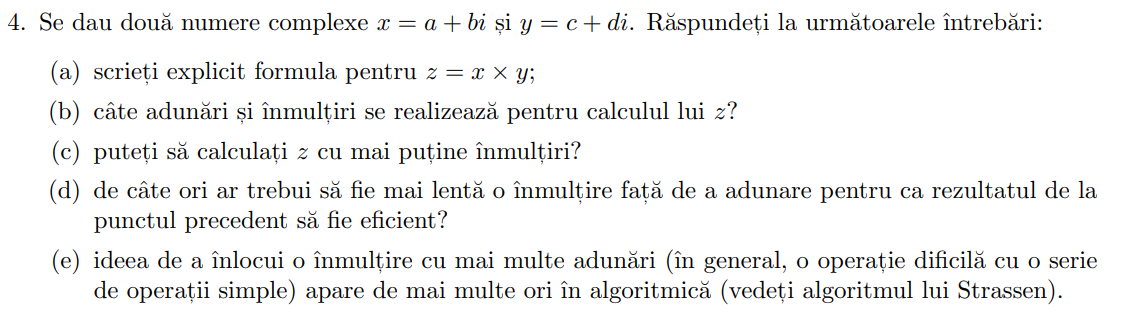
Ne interesează să putem răspunde la cât mai multe cereri de date venite de pe Internet:  
- Utilizare CPU (Eventual sisteme multi-core), media aritmetică  
  
Vrem să ne asigurăm că cererile de date sunt finalizate în 10ms  
- Wall-clock time, media aritmetică  
  
Ne interesează să ocupe maxim 100MB în memorie  
- Memoria RAM, maximum  
  
Vrem să rulăm sistemul de calcul cu un cost cât mai mic  
- Perfomanța 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 perfomanța folosind minimul/maximul?  
- Minimul – Când zgomotul / erorile din sistem sunt minime (best case)  
- Maximum – Când zgomotul / erorile din sistem sunt maxime (worst case)

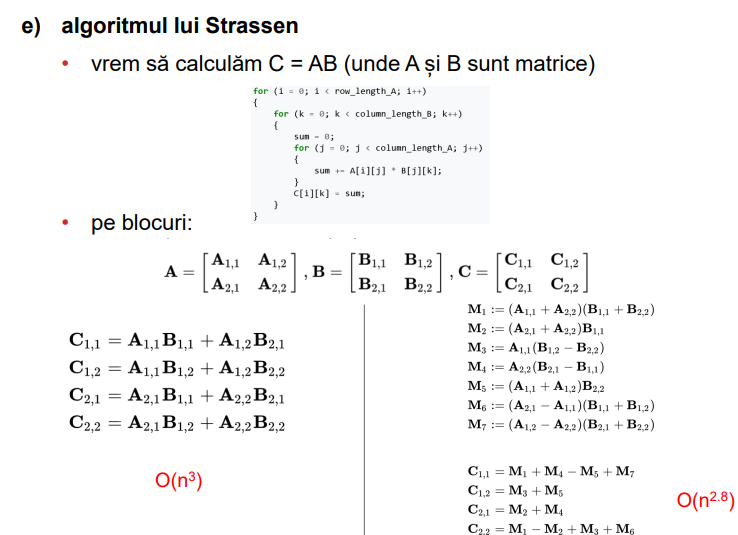
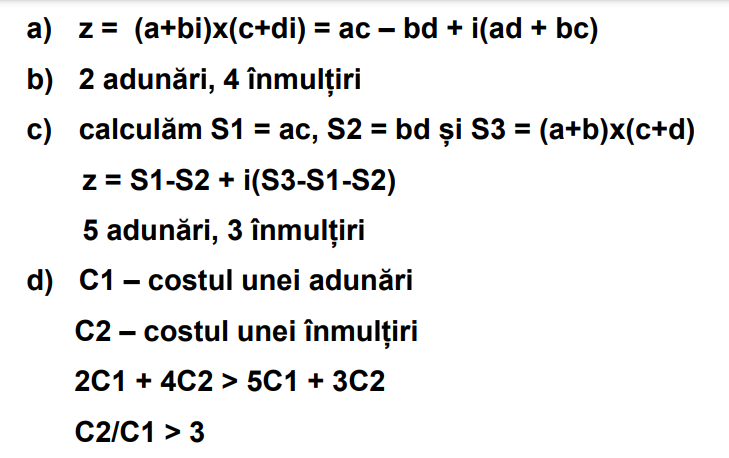
De ce este important să măsurăm cât mai bine / exact performanța unui sistem de calcul?  
- Dacă măsurăm cât mai bine și exact fiecare componentă, putem optimiza cât mai bine.

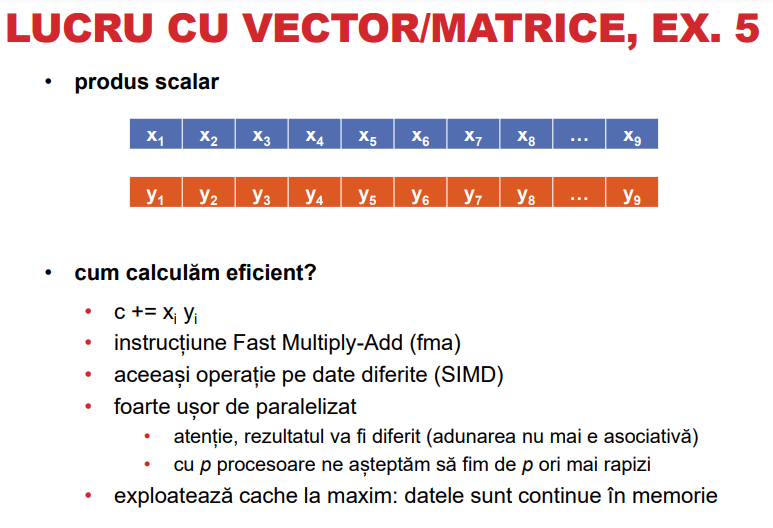


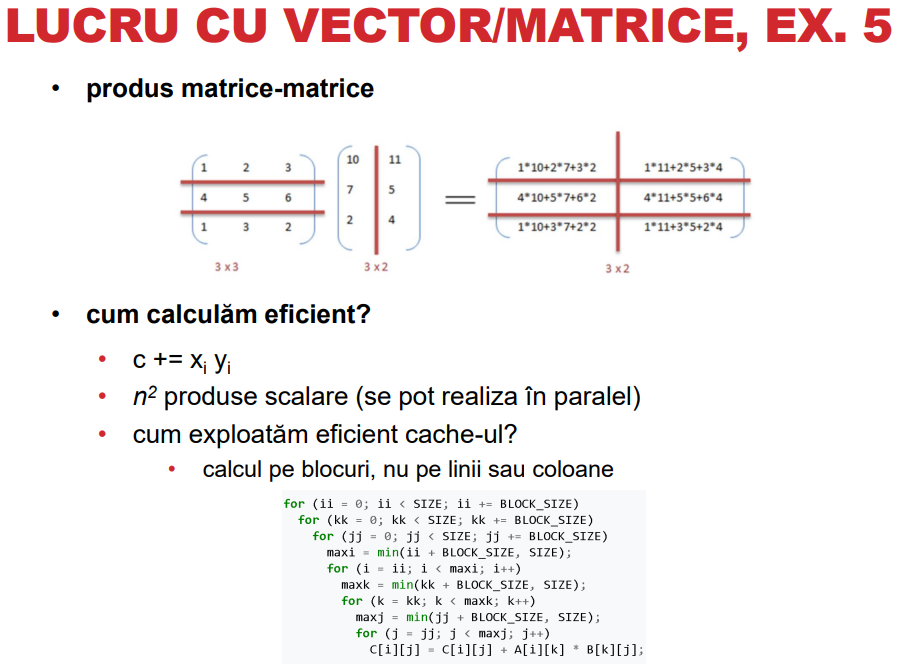
 

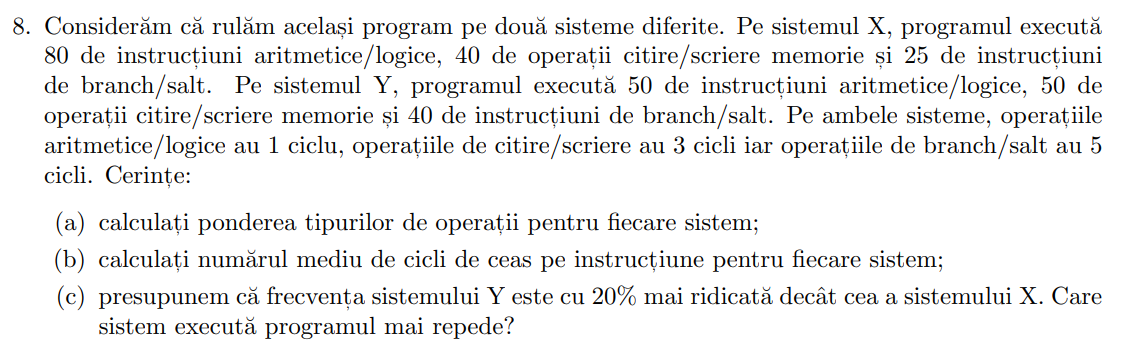


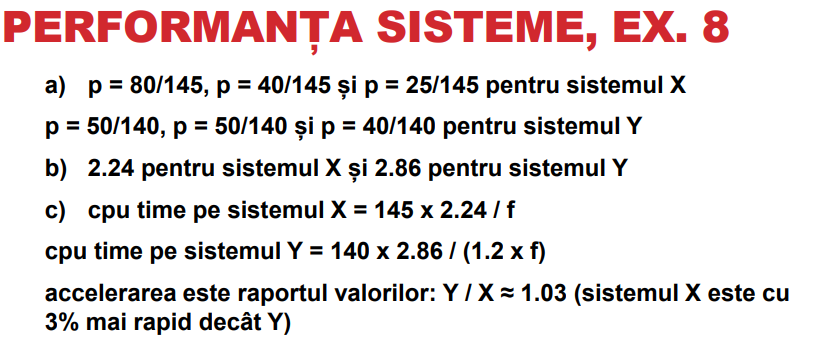












**PUTERILE LUI 2**

