**Cursul 1**

**-**inainte de al doilea razboi Mondial 50 op/sec (sua)

-1947-1955 sua, 1000 op/sec

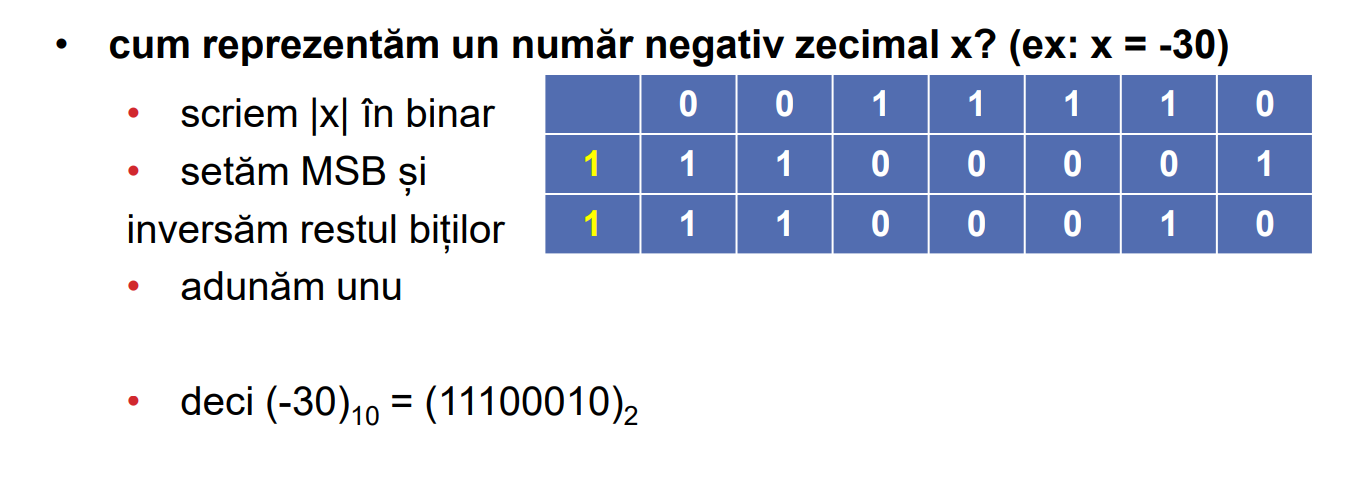
-2021 sua- 1685 peta op/sec

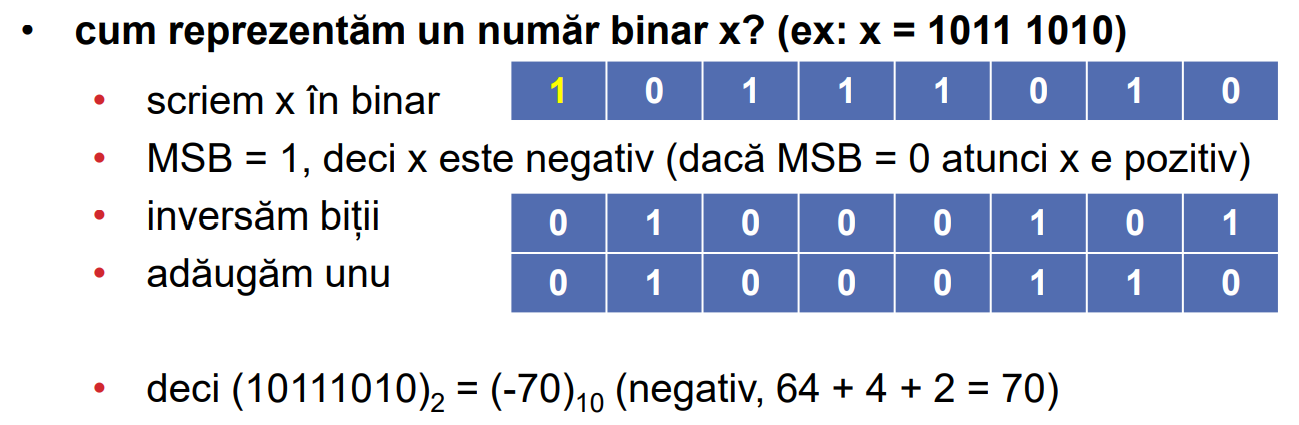
-ordinea in care se executa instructiunile este foarte importanta

**Cursul 2 – istoria, baze, complement fata de 2**

Contributii majore in istoria sist de calcul: Pascal, Leibniz, Boole, Babbage(“tatal calculatoarelor moderne”), Ada Lovelace(scrie primul program-calc nr Bernoulli), Turing, Shannon(parintele teoriei informatiei)

→in baza B reprezentarea este x=suma(biBi) cu i de la 0 pana la n-1(atatea cifre avem)





**Cursul 3 – informate, entropia lui Shannon, decodare Huffman, detectare erori**

→probabilitate mai mica inseamna ca primim mai multa informatie

Formula: pt o variabila aleatoare X care poate lua N valori distincte de la x1,…,xN, iar fiecare valoare distincta vine cu o probabilitate p1,…,pN

Informatia = I(xi)=log2(1/pi) (se masoara in biti)

P(eveniment A si eveniment B) = P(eveniment A) x P(eveniment B)

→valoarea medie de informatie primita despre o variabila X s.n. entropie

H(X)=E(I(X))= (suma din probabilitate\*informatia primita)

→entropia ne sune ca nu putem coda o variabila sub valoarea entropiei (H(x) este limita de compresie posibila)

→codare mai eficienta=dimensiune variabila a codului

→algoritmul lui Huffman; unele even apar mai des, deci primesc o codare mai scurta

→(**mod construire arbore)** se iau evenimentele cele mai imporbabile si se pun in arbore, se creeaza un nod cu noul eveniment format din cele 2, se compara cu probabilitatile celorlalte evenimente, si se aseaza in functie de egalitate, pe cealalta frunza, apoi se formeaza nod cu toate evenimentele asezate pana acum si se repeat.

→DISTANTA HAMMING=cati biti sunt diferiti intre cele doua siruri

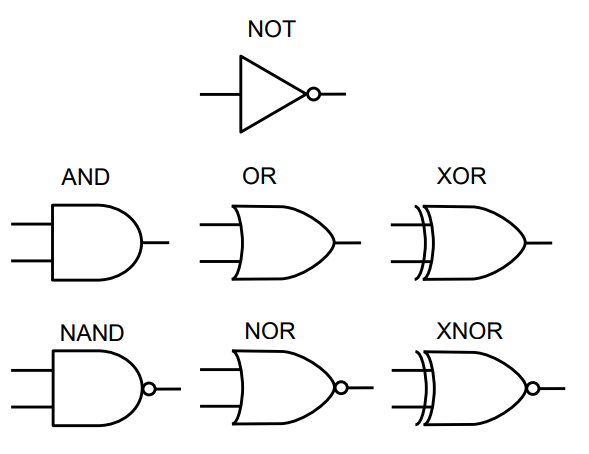
daca stocam doi biti in loc de unul(adaugam unul pt paritate), devenim redundanti, putem pierde cv si tot ne putem da seama de eroare, dar suntem de 2 ori mai ineficienti si nici nu putem detecta erori multiple.

→o distanta hamming de **2E+1** poate corecta E erori

→orice comunicatie/stocare este redundanta

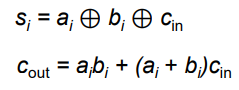
**Cursul 4 – abstractizare digitala, circuite**

**Porti logice:**



→ la baza tuturor se afla circuite electronice bazate de transistor

→la tabelele de adevar, vedem unde necunoscuta(de ex X) este 1, si apoi negam inputurile care au fost 0 si le inmultim cu inputurile care au fost 1, adunam pentru toate valorile de 1 ale variabilei si acela este rezultatul.

**Cursul 5 -porti logice sr latch, d latch, d flip flop**

→pt circuitul de adunare, si il folosim in cascada(serie)

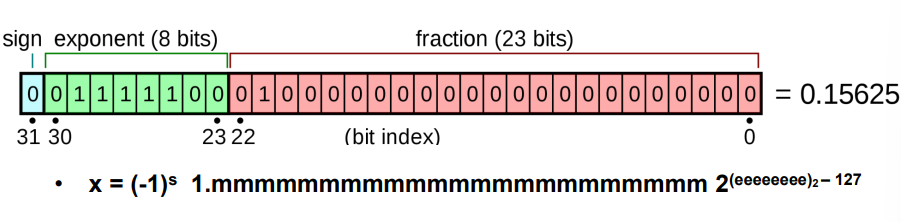
→Set-Reset Latch memoreaza un bit de informatie

**Cursul 6 – inmultire, impartire, Floating Point**

→pentru inmultirea numerelor in complement fata de 2, trebuie sa extindem operanzii pe 8 biti si facem inmultirea obisnuita. (primim rezultatul pe 8 biti)

→reprezentarea stiintifica 5024=5.024x103

→pentru reprezentarea in Floating Point, virgula mobile (m=mantisa,rosu)



**Cursul 7**

**→**CPU porneste BIOS

→cpu comunica cu perifericele si memoriile prin magistrala(BUS), acestea ajung la bus prin intermediul controlerlor.

→calculatoarele recente incep sa nu mai fie Von Neumann, adica au ALU, si memorie separata pentru cod si pentru date

→instructiunile(UC): IF(Instruction Fetch), ID(Instruction Decode), EX(Execute), MEM(Memory Access), WB(Write Back)

→SSD Solid State Disk -memorie flash rapida, azi scumpa, scriere mai lenta decat citirea

→HDD Hard Disks -mecanic

→deci comunicarea e la fel de importanta ca si procesarea in sine(comunicarea intre memorii, de la memorie la cpu etc)

**Cursul 8 -cod sursa la cod masina, Instruction Set Architecture (ISA)**

→comunicarea cu perifericele se face, de obicei, prin buffer-e in memorie

→**cod masina**: instructiuni binare executate direct de CPU, se obtine din cod sursa(este generic). Codul masina e specific pt Assembler, CPU, OS.

Cod sursa **→compilare** →cod assembly **→ asamblare** → fisier obiect **→linkare+biblioteci externe** →fisier binar **→ load/executie** →proces

YMM (pentru AVX) XMM (pentru SSE) registrii pt operatii pe vectori

→exceptia de la regula o face bytecode ul: instructiunile sunt executate de un interpretor care apoi le trimite la CPU(Java, C#, Python, Javascript)

**Cursul 9- pipelining, branch prediction, out of order execution**

→CPU executa IF, ID, EX, MEM, WB

→Pipelining este un tip de paralelism la nivel de instructiune(Instruction Level Paralellism – ILP)

→**Posibile hazards:** Structural hazards: doua instr incearca sa acceseze aceeasi unitate; Data Hazards: o instr depinde de rezultatul unei instr precedente; Control Hazards: din cauza unor jmp nu stim instr urmatoare

→pt instructiuni “dificile” avem registri speciali (xmm)

→in general predictia este binara, daca sari sau nu (then/else)

**Predictia cu istoric, predictie fixa, predictia de la pasul anterior, executie speculative(eager execution – calc cu si fara salt)**

**Cursul 10 -tipuri de parallelism, ierarhizarea memoriei, cache**

→rularea simultana a programelor este “simulata” (daca exista un singur core)

→daca am avea multe core-uri ar fi bine pentru rularea mai multor programe insa apar probleme, toate comunica pe acelasi bus, toate vor instructiuni, toate vor sa acceseze memoria. Solutia este ca fiecare core sa aiba o “memorie” locala -IERARHIZAREA MEMORIEI

**Tipuri de parallelism:**

1. **La nivel de biti –** modificarea dimensiunii “cuvintelor” procesorului (16,32,64 biti)
2. **La nivel de instructiune**  - pipelines
3. **La nivel de task-uri –** multi-thread, multi**-**process
4. **La nivel de blocuri** – vectorizarea operatiilor, operatii pe blocuri
5. **In trecut 1 procesor= 1 unitate calcul, de cateva decenii 1 procesor= 2,4,… unitati de calcul(sisteme multi-core)**

**→**sistemele multi-core au proprii: registrii, ALU, FPU, cache

Avantaje: coerenta cache-ului

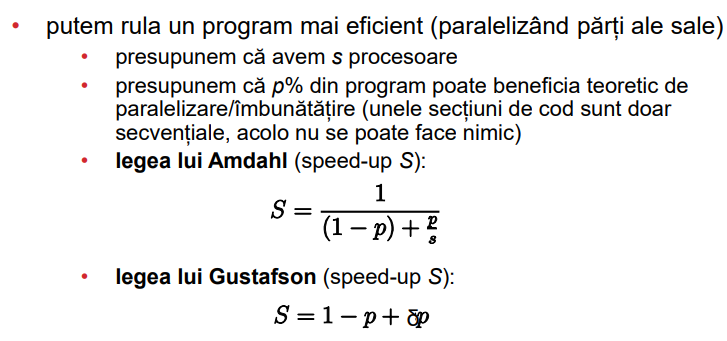
Dezavantaje: software special

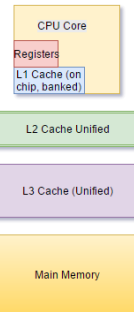
→sistemele multi-thread: Hyper-Threading, Chip Multi-threading au ca resurse proprii doar registrii.

Avantaje:daca resursa este disponibila

Dezavantaje: competitie pt resurse

Hyper-threading = logical cores

 !atentie simbol dubios=delta

**Tipuri de memorie:**

* Registrii procesorului: acces imediat, 100-1000 bytes
* Cache L0: acces foarte rapid, 5-20 kbytes
* Cache L1 (cache instr si date): 700GB/s, 100-500 kbytes
* Cache L2: 200GB/s, 500-1000 kbytes
* Cache L3 (de obicei partajat): 100 GB/s, 1-5 MB
* Memoria principala RAM: 100-500 MB/s, 2-64 GB
* Disc HD/SSD: 10-100 MB/s, 1TB

→ miss rate in RAM 0% in RAM sigur avem informatia

UMA(Uniform Memory Access) procese diferite pe procesoare diferitem dar e nev de modificarea programelor ca acestea sa ruleze paralel

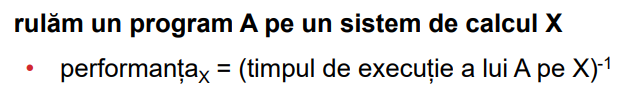
Dezavantaj: cache coherence

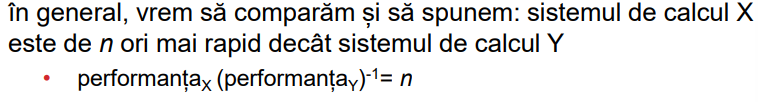
Asymmetric multiprocess systems: fiecare processor se ocupa de cv diferit

**Cursul 11-CPU, performanta calculatoare, multi-core, CISC vs RISC, consum de energie**

**→**GPU: unitate de calc separata, conectata la BUS, complementeaza capac de calc a unui CPU, contin un nr mare de unitati de procesare separate, specializati pe un anumit set de instructiuni, are propria sa memorie(on chip)

**DEF:**performanta= timp de executie program



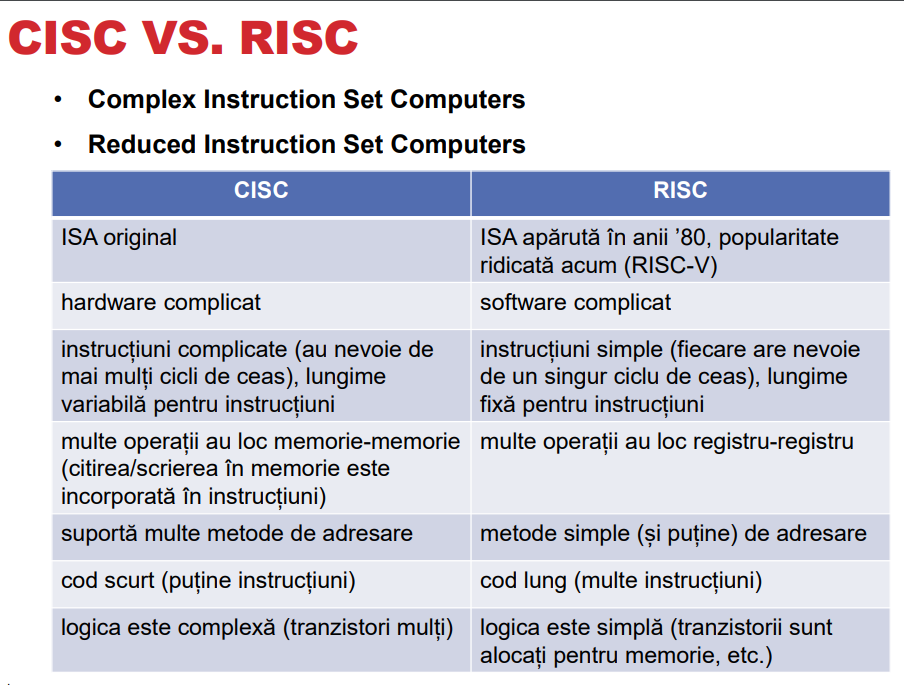


→timp de exec mai mic => performanta mai mare

**Ciclii de ceas pentru A = numar instr in A x numar ciclii necesar pt a executa o instr(in medie)**

**CPU time pentru A = (**formula anterioara)/**frecventa**

**Putere=O(tensiune2 x frecventa)**



**CISC**(Complex Instruction Set Computers): x86, Motorola

**RISC**(Reduced Instruction Set Computers): MIPS (Microprocessor without Interlocked Pipelined Stages), Power PC, Atmel AVR (masini Harvard), PIC, Microchip, ARM (Advanced RISC Machine), RISC-V

→Criteriu nou de performanta: cantitate de energie consumata- ff important pt dispozitive pe baterie (aceste dispozitive determina dinamic ciclul de ceas pt a balansa consume de energie => RISC domina CISC aici

**Cursul 12**

**→**Fisierele binare elf- header+continut(segmnete, sectiuni si instr/date)

Executie:

→SYSCALL pt exec = EXEC

→se citeste header-ul fisierului

→toate directivele LOAD sunt executate

→executia e preluata de *entry point address* (\_start si apoi main())

→simbolurile functiilor pot fi scoase cu strip

Static linking: simb din bibl ext sunt incluse in binar la link-are

Dynamic linking: link-uri la simboluri din bibl ext sunt adaugate la link-are, iar la rulare loader-ul rezolva aceste link-uri

Realizat de dynamic linker

Adresele simbolurilor sunt calculate prin binding:

* Cand programul este executat immediate binding
* Cand simb e prima data folosit- lazy binding

→bibliotecile sunt de 2 feluri: statice(adaugata la compilare), dinamice/commune(link-ata in timp real la exec)

→fiecare proces acceseaza adrese virtuale (logice), nu adrese fizice!

→adresele virtuale sunt translatate in adrese fizice(de asta se ocupa SO, kernel-ul)

→TLB:Translation lookaside buffer- cache ca traducerea sa se faca rapid

**Alinierea memoriei**

→CPU citeste in blocuri si vrea ca blocurile sa fie aliniate

Start=adresa de start ; align=alinierea pe care o vrem(putere a lui 2)

