

## Curs 1 – Seminar 1 – Baze – (Yey)

$0_{hex}$	=	$0_{dec}$	=	$0_{oct}$	0	0	0	0
$1_{hex}$	=	$1_{dec}$	=	$1_{oct}$	0	0	0	1
$2_{hex}$	=	$2_{dec}$	=	$2_{oct}$	0	0	1	0
$3_{hex}$	=	$3_{dec}$	=	$3_{oct}$	0	0	1	1
$4_{hex}$	=	$4_{dec}$	=	$4_{oct}$	0	1	0	0
$5_{hex}$	=	$5_{dec}$	=	$5_{oct}$	0	1	0	1
$6_{hex}$	=	$6_{dec}$	=	$6_{oct}$	0	1	1	0
$7_{hex}$	=	$7_{dec}$	=	$7_{oct}$	0	1	1	1
$8_{hex}$	=	$8_{dec}$	=	$10_{oct}$	1	0	0	0
$9_{hex}$	=	$9_{dec}$	=	$11_{oct}$	1	0	0	1
$A_{hex}$	=	$10_{dec}$	=	$12_{oct}$	1	0	1	0
$B_{hex}$	=	$11_{dec}$	=	$13_{oct}$	1	0	1	1
$C_{hex}$	=	$12_{dec}$	=	$14_{oct}$	1	1	0	0
$D_{hex}$	=	$13_{dec}$	=	$15_{oct}$	1	1	0	1
$E_{hex}$	=	$14_{dec}$	=	$16_{oct}$	1	1	1	0
$F_{hex}$	=	$15_{dec}$	=	$17_{oct}$	1	1	1	1

X	Y	X AND Y	X	Y	X OR Y	X	Y	X XOR Y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

- cum reprezentăm un număr binar  $x$ ? (ex:  $x = 1011 1010$ )

- scriem  $x$  în binar
  - MSB = 1, deci  $x$  este negativ (dacă MSB = 0 atunci  $x$  e pozitiv)
  - inversăm biți
  - adăugăm unu
- |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |

- cum reprezentăm un număr negativ zecimal  $x$ ? (ex:  $x = -30$ )

- scriem  $|x|$  în binar
  - setăm MSB și inversăm restul bițiilor
  - adunăm unu
- |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |

a)  $2^N - 1$

b)  $2^{N-1} - 1$  și  $-2^{N-1}$

c) aproximativ  $\log_2 x$ , exact sunt  $\text{ceil}(\log_2(x+1))$

d)  $4k$

e)  $\text{ceil}(k/4)$

f)  $\text{ceil}(k \log_2 10)$

## • entropia

- valoarea medie de informație primă 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

- care sunt avantajele codării cu dimensiune fixă?

- e simplu, știm că fiecare simbol are același număr de biți (deci știm de cât spațiu avem nevoie, etc.)
- putem accesa direct al i-lea simbol din sir (ABBA etc.)
- implementarea în circuite electronice se face la fel pentru că toate elementele au același număr de biți (32 sau 64 de biți)
- este optimă dacă simbolurile au probabilități egale

- care sunt avantajele codării cu dimensiune variabilă?

- codare eficientă (spațiu de stocare)

- primul pas:

- trebuie să definim o distanță între sirul corect și cel corrupt
- distanța Hamming între două siruri binare: câtă biți sunt diferenți (biți de pe aceleași poziții în prezentarea binară)
- sirurile trebuie să aibă aceeași lungime
- distanța Hamming mai sus: 3

- o distanță Hamming de  $2E+1$  poate corecta  $E$  erori

- cu aproximarea lui Stirling:  $\log_2(52!) \approx 52\log_2(52) - 52\log_2 e = 221.4$  biți

a)  $p = C_2^N / 2^N$

b)  $p = C_{N/2}^N / 2^N$ , presupunem  $N$  par

c)  $p = (\%N + 1) / 2^N$ , presupunem  $N$  divizibil la 4

d)  $p = 2^{N-1} / 2^N = 1 / 2$

e)  $p = 2^{N-1} / 2^N = 1 / 2$

f)  $p = N / 2^N$

g)  $p = 2^{N/2} / 2^N = 1 / 2^{N/2}$ , presupunem  $N$  par

h)  $p \approx [(2^N - 1) / \ln(2^N - 1)] / 2^N \approx [2^N / \ln(2^N)] / 2^N = 1 / \ln(2^N)$

i)  $p = (1 + \sum_{l=1}^{N/2} C_{2l}^N) / 2^N = 1 / 2$ , presupunem  $N$  par

j)  $p = 1 / 2^N$

## Curs 2 – Seminar 2 – Entropie/Informație (Ok)

- introducem o variabilă aleatoare  $X$ :

- această variabilă poate lua  $N$  valori distincte ( $x_1, x_2, \dots, x_N$ )
- fiecare valoare distinctă apare cu probabilitate ( $p_1, p_2, \dots, p_N$ )

- câtă informație primim dacă observăm că variabila  $X$  a luat valoarea  $x_i$ ?

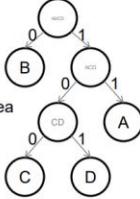
$$I(x_i) = \log_2 \left( \frac{1}{p_i} \right)$$

- algoritmul Huffman

- input: probabilitatea fiecărui eveniment {1/3, 1/2, 1/12, 1/12}

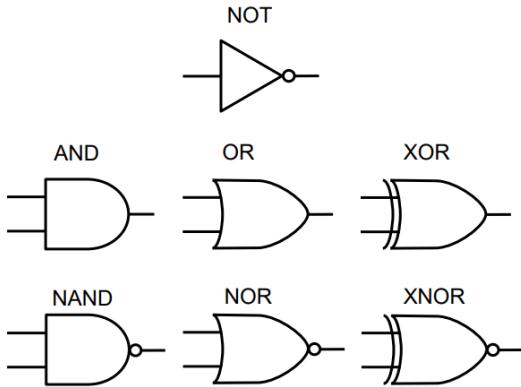
- algoritmul:

- luăți evenimentele cele mai improbabile
  - C și D, și le punem în arbore
  - creăm un nou eveniment CD, probabilitatea de apariție a acestuia este 1/6 (suma C și D)
  - noile evenimente sunt {A, B, CD} cu probabilități {1/3, 1/2, 1/6}
- din nou evenimentele cele mai improbabile
  - A și CD, A merge pe cealaltă frunză
  - noile evenimente sunt {B, ACD} cu probabilități {1/2, 1/2}
- din nou evenimentele cele mai improbabile
  - acum sunt doar două, {B, ACD}, B merge pe cealaltă frunză

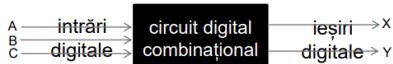


- Se dă un număr natural  $x$  pe  $N$  biți. Câtă informație am câștigat dacă:

- ni se spune despre  $x$  că are exact două valori "1" în reprezentarea sa binară;
- ni se spune despre  $x$  că are exact  $N/2$  valori "1" în reprezentarea sa binară;
- ni se spune despre  $x$  că are o secvență continuă de  $N/4$  biți de "1" în reprezentarea sa binară (restul bițiilor sunt "0");
- ni se spune despre  $x$  că are MSB setat la "1";
- ni se spune despre  $x$  că este impar;
- ni se spune despre  $x$  că este o putere a lui 2;
- ni se spune despre  $x$  că are primii  $N/2$  biți din reprezentarea sa binară setați la "0";
- ni se spune despre  $x$  că este un număr prim (aici doar o estimare aproximativă este posibilă);
- ni se spune despre  $x$  că are în reprezentarea sa binară un număr par de biți setați la "1";
- ni se spune că  $x = 42$ .



• circuit digital combinațional



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

• o expresie booleană care conține regulile din tabel?

- $X = \overline{ABC} + \overline{ABC} + \overline{BC} + ABC$   
 $= \overline{A}(\overline{B}C + B\overline{C}) + A(\overline{B}\overline{C} + BC)$   
 $= \overline{A}(B \oplus C) + A(B \oplus C)$   
 $= A \oplus B \oplus C$
- $X = AC + BC + AD + BD$   
 $= (A + B)C + (A + B)D$   
 $= (A + B)(C + D)$

• două întrebări importante:

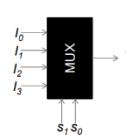
- de ce folosim semnale digitale în loc de analogice?
  - din cauza zgomotului
  - într-un sistem analogic zgomotul se acumulează
  - într-un sistem digital, avem corecțiile de zgomot (avem margini)
- de ce folosim sistemul binar? ar fi mai avantajos să folosim hex?
  - da, ar fi mai avantajos să folosim hex (e de 4 ori mai avantajos)
  - problema este că în loc de două stări ar trebui acum să avem 16
  - asta înseamnă că trebuie să distingem 16 nivele de voltaj în prezența zgomotului (adică cu tot cu margini de zgomot)
  - probabil 16 nivele e prea mult ... dar probabil 4 nivele ar fi fezabil
  - dacă am avea 4 nivele (adică baza  $B = 4$ ) am fi de două ori mai eficienți

$$A \oplus B = \overline{AB} + A\overline{B}$$

Legile de Morgan sunt:  $\overline{A+B} = \bar{A}\bar{B}$  și  $\overline{AB} = \bar{A} + \bar{B}$ .

• multiplexare

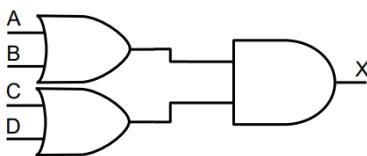
- un circuit care selectează un semnal digital de la intrare pe baza unui semnal de activare s



$s_1$	$s_0$	$Y$
0	0	$I_0$
0	1	$I_1$
1	0	$I_2$
1	1	$I_3$

- de ce folosim acest circuit?
  - implementare hardware pentru "if" și "case"
  - implementare hardware pentru operații shift

- $\overline{A+B}$
- $\overline{\bar{A}\bar{B}}$
- $\overline{A+B+C}$ , generalizati la  $N$  variabile digitale  $A_i$
- $\overline{ABC}$ , generalizati la  $N$  variabile digitale  $A_i$
- $(A+B)\bar{A}\bar{B}$
- $\overline{AB}(\bar{A}+\bar{B})$
- $(A+B)(\bar{A}+\bar{B})$
- $\overline{AB}(\overline{AB})$
- $C + \overline{CB}$
- $\overline{AB}(\bar{A}+B)(\bar{B}+\bar{B})$
- $(\overline{AB})(\bar{B}+C)$
- $(\overline{A+B})(\bar{B}C)$
- $(\bar{A}+C)(\overline{AB})$

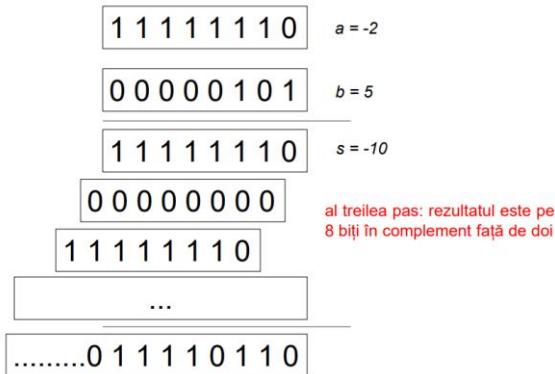


- $A + 0$
- $\bar{A}0$
- $A + \bar{A}$
- $A + A$
- $A + AB$
- $A + \bar{A}B$
- $A(\bar{A} + B)$
- $AB + \bar{A}B$
- $(\bar{A} + \bar{B})(\bar{A} + B)$
- $A(A + B + C + \dots)$
- fie  $f(A, B, C) = A + B + C$ 
  - $f(A, B, AB)$
  - $f(A, B, \bar{A}\bar{B})$
  - $f(A, B, \overline{AB})$
- $A + A\bar{A}$
- $AB + A\bar{B}$

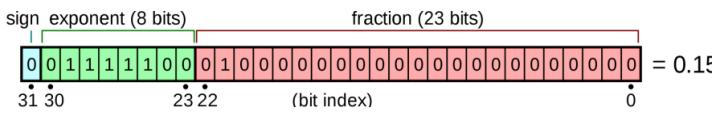
- $\bar{A} + B\bar{A}$
- $(D + \bar{A} + B + \bar{C})B$
- $(A + \bar{B})(A + B)$
- $C(C + CD)$
- $A(A + AB)$
- $\overline{A + A}$
- $\overline{A + A}$
- $D + (D\bar{C}BA)$
- $\bar{D}(\overline{DBCA})$
- $AC + \bar{A}B + BC$
- $(A + C)(\bar{A} + B)(B + C)$
- $\bar{A} + \bar{B} + AB\bar{C}$
- $(A + C)(AD + A\bar{D}) + AC + C$
- $\bar{A}(A + B) + (B + AA)(A + \bar{B})$
- $(A + B)^2 + (A + B)^3 + A + 3\bar{A} + A^3$

- $A + 0 = A$
- $\overline{Ax0} = 0$
- $A + \overline{A} = 1$
- $A + A = A$
- $A + AB = A$
- $A + \overline{AB} = A + B$
- $A(\overline{A} + B) = AB$
- $AB + \overline{AB} = B$
- $(\overline{A} + \overline{B} + AB) = \overline{A}$
- $A(A + B + C + \dots) = A$
- subpuncte
  - $A + B$
  - 1
  - 1
- $A + \overline{A}A = A$
- $AB + A\overline{B} = A$
- $\overline{A} + B\overline{A} = \overline{A}$
- $C(C + CD) = C$
- $A(A + AB) = A$
- $\overline{(A + \overline{A})} = A$
- $\overline{!(A + \overline{A})} = 0$
- $D + (D\bar{C}BA) = D$
- $IDI(DBCA) = ID$
- $AC + \overline{AB} + BC = AC + \overline{AB}$
- $(A + C)(\overline{A} + B)(B + C) = AB + \overline{AC}$
- $\overline{A} + B + AB\overline{C} = \overline{A} + \overline{B} + \overline{C}$
- $(A + B)^2 + (A + B)^3 + A + 3\bar{A} + A^3 = 1$

- exemplu,  $s = a \times b$



- standardul: IEEE 754 Floating Point



$$\bullet x = (-1)^s \cdot 1.\text{xxxxxxxxxxxxxx} \cdot 2^{(\text{eeeeeee})_2 - 127}$$

$a \times 2$

- soluția:  $a \ll 1$ , sau  $a + a$ 
  - soluția:  $a \& 0x000F$

$a \times 16$

- soluția:  $a \ll 4$ 
  - soluția:  $a \& 0xFFFF00$

$a \times 3$

- soluția:  $a \ll 1 + a$ 
  - împărțiți la 4
    - soluția:

$a \times 7$

- soluția:  $a \ll 3 - a$ 
  - vrem exponentul, unde se află?
    - MASK = 0x7F800000
    - extragem exponent =  $(a \& MASK) \gg 23$
    - dacă exponent > 1 atunci exponent = exponent - 2, altfel a = 0
    - trebuie să actualizăm a
      - $a = (a \& \sim MASK) | (\text{exponent} \ll 23)$

$a / 8$

- soluția:  $a \gg 3$

$a \bmod 16$

- soluția:  $a \& 0x000F$

$a \times 72$

- soluția:  $a \ll 6 + a \ll 3$

În 1991, în timpul războiului din Golf (operațiunea "Furtună în desert") sistemul de apărare Patriot al U.S. nu a reușit să intercepteze o rachetă SCUD a forțelor armate Irakiene. Intern, sistemul american avea o operație aritmetică unde făcea conversia din timp discret (număr întreg) în timp real (secunde, număr real) înmulțind timpul discret cu 0.1 (pentru a obține numărul de secunde). Sistemul avea 0.1 aproximativ pe 24 de biți astfel

$$(0.1)_{10} \approx (0.00011001100110011001100)_2.$$

Cerințe:

- calculați aproximarea binară de mai sus;
- cum este diferența dintre valoarea calculată și 0.1;
- cum este eroarea (de timp) după 100 de ore de operare;
- cum este eroarea (de timp) după 100 de ore de operare;
- cum este eroarea dacă reprezentăm 0.1 în formatul IEEE 754 FP?
- dacă rachetele SCUD pot atinge o viteza MACH 5, cum este distanța pe care racheta o poate parcurge în timpul eroare calculat?
- ce faceți ca să corectați problema?

#### • care este problema cu această reprezentare?

- partea întreagă este separată de partea fraționară
- fiecare are nevoie de un număr de biți prestatibil
- asta poate să fie ineficient
- vrem ca numărul de biți total să fie alocat "dinamic", în funcție de numărul pe care trebuie să îl reprezentăm

#### • când trebuie să reprezentăm un număr real

- nu putem să avem precizie infinită
- avem un număr finit de biți, deci putem să scriem biții în circuite
- avem nevoie de precizie variabilă
- putem avea precizie "infinită" dacă avem numere raționale (și vom salva separat numărătorul și numitor ca întregi)

#### • schimbați semnul lui a

- soluția:  $a = a \wedge (1 \ll 31)$

#### • calculăm $\text{abs}(a)$

- soluția:  $a = a \wedge \sim(1 \ll 31)$

$$\bullet -1313.3125$$

- partea întreagă este: 1313
- partea fraționară: 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 01001000010101000000000
- exponentul este  $10 + 127 = 137 = 10001001_2$
- semnul este 1

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

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

#### calculați aproximarea binară

- soluția: 0.09999904632568359375

#### care este diferența dintre valoarea calculată și 0.1

- soluția:  $0.1 - 0.09999904632568359375$

#### cum este eroarea (de timp) după 100 de ore de operare

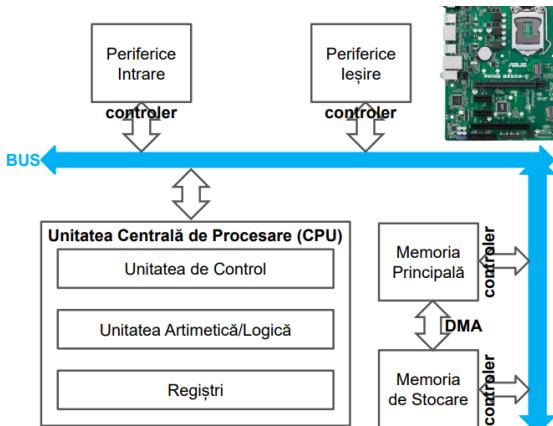
- soluția:  $100 \times 60 \times 60 \times 10 \times (0.1 - 0.09999904632568359375) \approx 0.34$

#### cum este eroarea dacă reprezentăm 0.1 în formatul IEEE 754 FP?

- soluția:  $100 \times 60 \times 60 \times 10 \times (0.1 - 0.0999999403953552) \approx 0.021$

#### dacă rachetele SCUD pot atinge o viteza MACH 5, cum este distanța pe care racheta o poate parcurge în timpul eroare calculat?

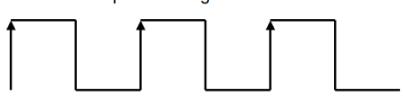
- soluția:  $1715 \text{ m/s} \times 0.34 \text{ s} \approx 583 \text{ m}, 1715 \text{ m/s} \times 0.021 \text{ s} \approx 36 \text{ m}$



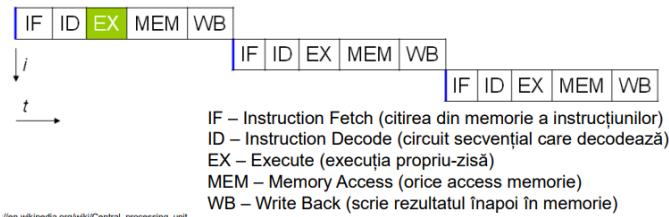
- 5 componente principale:
  - Clock
  - regiștri (“memoria”)
  - UAL (“operații”)
  - BUS
  - UC (“instructiunile”)

#### • Clock

- este un circuit special care generează “ceasul”

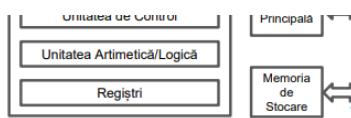


- este frecvența la care operează (calcule și sincronizarea componentelor secvențiale) CPU-ul
- cu cât este mai mare frecvența, cu atât mai bine (în general)
- se măsoară în MHz sau GHz



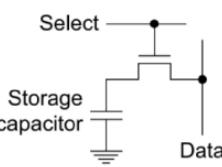
#### • Static RAM (SRAM)

- bazată pe flip-flops
- rapid
- scump
- regiștrii din CPU sunt de același tip



#### • Dynamic RAM (DRAM)

- fiecare bit este reprezentat de o combinație tranzistor + condensator
- condensatoarele suferă de leakage (scurgeri de tensiune)
- DRAM trebuie actualizat o dată la fiecare câteva zeci de ms



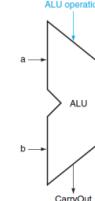
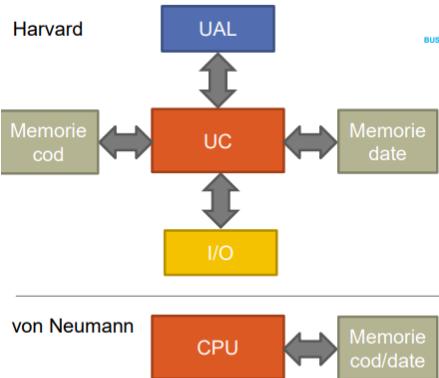
#### FLAGS

**Instruction Pointer (IP):** următoarea instrucțiune care trebuie executată

**Stack Pointer (ESP):** adresa stivei

**YMM (pentru AVX) / XMM (pentru SSE):** regiștri pentru operații pe vectori

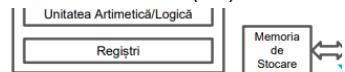
#### • Tipul arhitecturii de calcul



după orice operație, stim "gratuit" dacă rezultatul a fost sau nu zero – este folositor?

#### • DDR RAM

- Double Data Rate RAM
- .../DDR4/DDR5/DDR6
- performanța este definită de:
  - capacitate
  - dacă au un sistem intern de corectarea erorilor (ECC)
  - tempi de acces (în cât timp de la comanda de citire de biți din RAM avem datele disponibile?, timpul de refresh)
  - consumul de energie

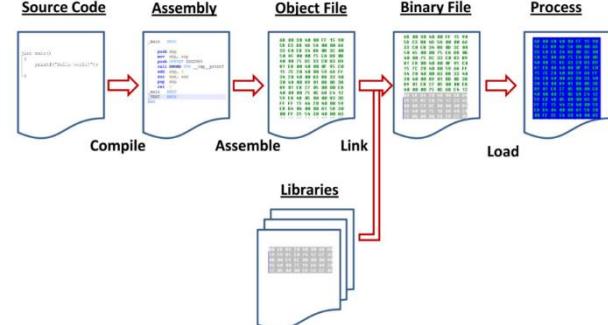


#### • SSD (Solid State Disks)

- e memorie flash, rapidă
- azi, e scumpă
- scrierea e mult mai lentă decât citirea

#### • HDD (Hard Disks)

- mecanic

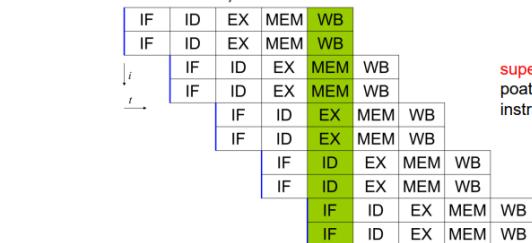


#### • Instruction Set Architecture (ISA)

- structura sintactică și semantică a limbajului Assembly
  - regiștri
  - instrucțiuni
  - tipuri de date
  - metode de adresare a memoriei

4 bits = 1 nibble  
8 bits = 1 byte  
16 bits = 1 word  
32 bits = 1 dword  
64 bits = 1 qword

- adresare imediată:**
  - imediat: mov \$172, %rdi
  - cu registru: mov %rcx, %rdi
  - cu memorie: mov 0x172, %rdi
- adresare indirectă**
  - indirect prin registru: mov (%rax), %rdi
  - indirect indexat: mov 172(%rax), %rdi
  - indirect bazat pe IP: mov 172(%rip), %rdi
- cazul cel mai general:** mov 172(%rdi, %rdx, 8), %rax
  - Base + Index\*Scale + Displacement
  - îl aveți explicat detaliat în suportul de laborator
- structural hazards**
  - două instrucțiuni încearcă să acceseze aceeași unitate



ia.org/wiki/Central\_processing\_unit

**superscalar**, hardware dublat, poate termina mai mult de o instrucție într-un ciclu

#### data hazards

- o instrucție depinde de rezultatul unei instrucții 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

#### ce se întâmplă în procesoarele moderne?

- se citesc câteva sute de instrucții la un moment dat
- în hardware, se realizează un graf (*data-flow graph*) de dependențe între aceste instrucții
- exemplu: quad(a, b, c)
  - t1 = a\*c; t3 = b\*t1; t6 = -b; t9 = 2\*t1;
  - 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

Presupunem că avem un sistem de calcul pe 32 de biți, răspundeți la următoarele întrebări scurte:

- care este adresa de memorie cea mai mare care poate fi accesată? (cu 8 biți / locație memorie)
- avem instrucția *jne etichetă*, unde *jne* are opcode-ul 0110. Care este saltul maxim care se poate realiza cu această instrucție?
- avem o instrucție *add R1, R2*, unde *add* are opcode-ul 0011 iar *R1* și *R2* sunt registri iar calculul realizat este *R2 ← R1 + R2*. Căți registri diferiți putem avea?
- similar cu instrucția anterioară, dar acum avem *add R1, R2, R3*, unde *add* are opcode-ul 0100 iar calculul realizat este *R3 ← R1 + R2*. Căți registri diferiți putem avea?

- a) **adresa de memorie cea mai mare accesibilă este  $2^{32} = 4\text{GB}$  (4.294.967.296 bytes)**

- b) **instrucția este *jne etichetă*, unde *jne* are opcode 0110**
- adresa de memorie cea mai mare accesibilă este  $2^{28} = 0.25\text{ GB}$

- c) **avem instrucția *add R1, R2***
- opcode este 0011
  - operăția suportă  $2^{14} = 16384$  registrii diferiți

- d) **avem instrucția *add R1, R2, R3***
- opcode este 0100
  - vom avea 9.33 biți pentru fiecare reprezentare a unui registru: două poziții suportă  $2^9 = 512$  iar o poziție  $2^{10} = 1024$

#### Categorii de instrucții

- transferul datelor:** mov, cmov, movq, movs, movz, push, pop
- aritmetică și logică:** add, sub, mul, imul, div, idiv, sal, sar, shl, shr, and, or, not, xor, test, cmp
- controlul programului:** call, ret, j\*

#### trei mecanisme pentru execuție cu viteză sporită

- pipelining (conductă de date)
- branch prediction (predictia salturilor)
- out of order execution (execuția în ordine arbitrară)



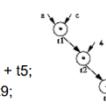
#### • imaginea arată bine, din păcate nu putem face aşa ceva mereu

- pipeline stalls (întârzieri în conductă de date)
- aceste evenimente se numesc hazards (erori)
  - structural hazards:** o unitate de calcul este deja utilizată
  - data hazards:** datele nu sunt pregătite pentru utilizare
  - control hazards:** nu știm următoarea instrucție
  - din cauza unor instrucții de jump nu știm instrucția următoare

#### 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 cu atât mai rău

- soluția: trebuie înlocuit hardware-ul
- soluția: sistemul de operare trebuie să ia în considerare problema totul va fi mai lent

##### out of order execution

- %eax ← %ebx + %ecx**  
**%eax ← %ebx + %edx** **WAW**
- %ebx ← %ecx + %eax**  
**%eax ← %edx + %eax** **WAR**
- %eax ← %ebx + %ecx**  
**%edx ← %eax + %edx** **RAW**
- %eax ← 6**  
**%eax ← 3** **WAW**  
**%ebx ← %eax + 7** **RAW , rezultatul poate fi 10 sau 13**

## Instruction-level parallelism (ILP)

- Instruction Pipelining
- Register Renaming
- Speculative Execution
- Branch Prediction
- Value Prediction
- Memory Dependence Prediction
- Cache Latency Prediction
- Out-of-order Execution
- Dataflow Analysis/Execution

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 bucătă de memorie, cât ne costă ca timp dacă:

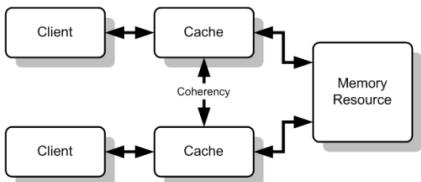
- verificăm în RAM: 50 ns
- verificăm în L1: 1 ns + (0.1 x 50 ns) = 6 ns
- verificăm în L2: 1 ns + (0.1 x (5 ns + (0.01 x 50 ns))) = 1.55 ns
- verificăm în L3: 1 ns + (0.1 x (5 ns + (0.01 x (10 ns + (0.002 x 50 ns))))) = 1.5101 ns

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 legătură miss rate? cu dimensiunea memoriei

corespondența dintre locația din cache și locația din RAM

- la citire nu sunt niciodată probleme
- la scriere lucrurile se complică
  - rezultatul este scris în cache
  - și celelalte memorii trebuie să fie anunțate de noua valoare
    - L1/L2/L3/RAM etc.
- situația se complică și mai mult dacă sunt cache-uri diferite pentru fiecare core pe care îl avem: cache coherence (protocol pentru consistența tuturor cache-urilor)



ce se întâmplă dacă avem un sistem multi-core?

- putem rula mai multe programe simultan
  - același program, instanțe diferite
  - diferite programe
- putem rula un program mai eficient (paralelizând părți ale sale)
  - presupunem că avem s procesoare
  - presupunem că p% din program poate beneficia teoretic de paralelizare/îmbunătățire (unele secțiuni de cod sunt doar secvențiale, acolo nu se poate face nimic)
  - **legea lui Amdahl** (speed-up S):
 
$$S = \frac{1}{(1 - p) + \frac{p}{s}}$$

- **legea lui Gustafson** (speed-up S):
 
$$S = 1 - p + \delta p$$

- **principiul de localizare**

• presupunem că avem în RAM un vector de 1000 elemente



• cache L1 conține a<sub>4</sub> și următoarele elemente din vector



• cache L2 conține a<sub>0</sub> și mai multe elemente din vector



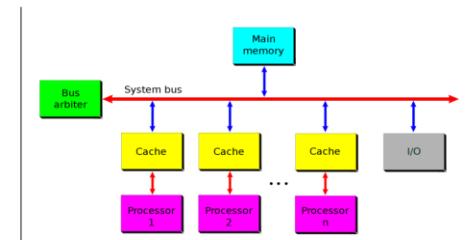
miss rate pentru cache L1? 25%

miss rate pentru cache L2? 10%

cine e responsabil de ce anume?

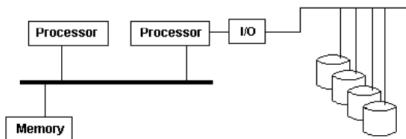
- programatorul: transfer între HD/SSD și RAM (citire de pe disc)
- logică hardware: din/în RAM în/din memoriile cache
- compilatorul: generează cod care exploatează cache-ul

## SYMMETRIC 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

## ASYMMETRIC MULTIPROCESS SYSTEMS



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

## GPU

- excelent pentru clasa de probleme “embarrassingly parallel”
  - operații matrice-vector, matrice-matrice
    - calculul transformatei Fourier
  - procesarea imaginilor
  - convolutional neural networks (CNN) pentru Machine Learning
  - căutări exhaustive (brute force search)
    - căutarea hyperparametrilor
  - crypto mining
  - integrare numerică
  - simulări fizice cu scenarii diferite (condiții inițiale diferite)
  - raytracing

$$\text{CPU time pentru A} = \frac{\text{număr instrucțiuni în A} \times \text{număr cicli necesar pentru a executa o instrucțiune (în medie)}}{\text{frecvența}}$$

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	deinde (20-80)
adunare floating point	addss, addsd	3
înmulțire floating point	mulss, mulsd	5
împărțire floating point	divss, divsd	deinde (20-80)
fused-multiply-add floating point	vfmass, vfmasd	5

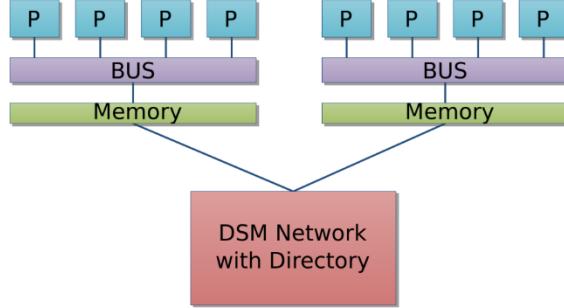
- ciclii de ceas pentru A = număr instrucțiuni în A × număr ciclii necesar pentru a executa o instrucțiune (în medie)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

- Complex Instruction Set Computers
- Reduced Instruction Set Computers

CISC	RISC
ISA original	ISA apărută în anii '80, popularitate ridicată acum (RISC-V)
hardware complicat	software complicat
instrucțiuni complicate (au nevoie de mai mulți cicli de ceas), lungime variabilă pentru instrucțiuni	instrucțiuni simple (fiecare are nevoie de un singur ciclu de ceas), lungime fixă pentru instrucțiuni
multe operații au loc memorie-memorie (citirea/scrierea în memorie este incorporată în instrucțiuni)	multe operații au loc registru-registrul
suportă multe metode de adresare	metode simple (și puține) de adresare
cod scurt (puține instrucțiuni)	cod lung (multe instrucțiuni)
logica este complexă (tranzistori mulți)	logica este simplă (tranzistorii sunt alocați pentru memorie, etc.)

## 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

rulăm un program A pe un sistem de calcul X

- performanța<sub>X</sub> = (timpul de execuție a lui A pe X)<sup>-1</sup>
- timp de execuție mai mic → performanță mai mare
- în general, vrem să comparăm și să spunem: sistemul de calcul X este de  $n$  ori mai rapid decât sistemul de calcul Y
  - performanța<sub>X</sub> (performanța<sub>Y</sub>)<sup>-1</sup> =  $n$

**CPU time pentru A = ciclii de ceas pentru A / frecvența**

- deci, putem micșora timpul de execuție pentru A dacă
  - reducem numărul de ciclii de ceas necesar pentru a executa A
  - mărim frecvența procesorului

### 1.2 Eight Great Ideas in Computer Architecture

- Design for Moore's Law
- Use Abstraction to Simplify Design
- Make the Common Case Fast
- Performance via Parallelism
- Performance via Pipelining
- Performance via Prediction
- Hierarchy of Memories
- Dependability via Redundancy

### Complex Instruction Set Computers

- x86
- Motorola

### Reduced Instruction Set Computers

- MIPS (Microprocessor without Interlocked Pipelined Stages)\*
- Power PC
- Atmel AVR (mașini Harvard)
- PIC Microchip
- ARM (Advanced RISC Machine)
- RISC-V

## 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)

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:

- calculați noua valoare  $m_{L1}$  pentru care timpul de acces la memoria este jumătatea  $t_{RAM}$ ;
- calculați noua valoare  $t_{L2}$  pentru care timpul de acces la memoria este o zecime din  $t_{RAM}$ ;
- calculați noua valoare  $m_{L3}$  pentru care timpul de acces la memoria este același  $t_{RAM}$ ;
- 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). Reduceți timpul de acces la memoria de o mie de ori față de  $t_{RAM}$  cu cost minim;

Presupunem că avem un sistem de calcul despre care vrem să știm anumite lucruri (legate de performanță sa). În scenarii diferite, ne interesează criterii de performanță diferite, fiecare având o metodă proprie de estimare. Rulăm același programe de mai multe ori și trebuie să decidem care este funcția de agregare (media aritmetică și geometrică, mediana, maximum și minimum). Ce criteriu de performanță am măsură și cum l-am estimă în următoarele cazuri:

- ne interesează să putem răspunde la căt mai multe cereri de date venite de pe Internet;
- vrem să ne asigurăm că cererile de date sunt finalizează în 10ms;
- ne interesează ca cererile să ocupe maxim 100MB în memorie;
- vrem să rulăm sistemul de calcul cu un cost căt mai mic;
- vrem să știm că majoritatea cererilor sunt servite în maximum 50ms;
- vrem să estimăm timpul total de răspuns al sistemului;
- este vreodată folositor să estimăm performanța folosind minimul/maximul?
- de ce este foarte important să măsurăm căt mai bine/exact performanța unui sistem de calcul?

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	(a) 7.25	(a) 6.75	(g) 1.57	(g) 0.64

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

Se dă 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 căte 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 algoritmiciă (vedeți algoritmul lui Strassen).

Există multe feluri de a calcula îmbunătățirea de performanță care este posibilă într-un sistem multi-core. Considerăm că avem un program care are  $0 \leq p \leq 1$  dintre instrucțiuni (interpretat ca procentaj) paralelizabile. Considerăm că avem la dispoziție  $s \geq 1$  procesoare sau că sistemul beneficiază de o accelerare a performanței de  $\delta$  ori. Două modalități de a calcula accelerarea execuției, legea lui Amdahl și legea lui Gustafson, respectiv:

$$S_{\text{Amdahl}} = \frac{1}{(1-p) + \frac{p}{s}} \quad \text{și} \quad S_{\text{Gustafson}} = 1 - p + \delta p. \quad (1)$$

Cerinte:

- calculați  $S_{\text{Amdahl}}$  și  $S_{\text{Gustafson}}$  pentru  $p = 0.5$ ,  $s = 8$  și  $\delta = 8$ ;
- încercați să deduceți voi legile lui Amdahl și Gustafson;
- ce se întâmplă cu  $S_{\text{Amdahl}}$  și  $S_{\text{Gustafson}}$  dacă  $p = 0$ ? Care este interpretarea rezultatului?

## Standard Performance Evaluation Corporation (SPEC)

- benchmarking standardizat
- evaluatează
  - performanța de calcul
  - consumul de energie

### 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

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

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

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

### 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\$

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

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

rezolvați pentru A, B și C

### a) utilizare CPU (eventual sisteme multi-core), media aritmetică

### b) wall-clock time, media aritmetică

### c) memoria RAM, maximum

### d) performanță per Watt, media aritmetică sau maximum

### e) wall-clock time, 50/90/99th percentile mediana

### f) wall-clock time speedup, media aritmetică

### g) minimum = când zgromotul/erorile din sistem sunt minime (best-case behavior), maximum = când zgromotul/erorile din sistem sunt maxime (worst-case behavior)

### h) dacă măsurăm că mai bine și exact fiecare componentă, putem optimiza că mai bine (semnificativ)

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

### b) 2 adunări, 4 înmulțiri

$$c) \text{calculăm } S_1 = ac, S_2 = bd \text{ și } S_3 = (a+b)x(c+d)$$

$$z = S_1 - S_2 + i(S_3 - S_1 - S_2)$$

### 5 adunări, 3 înmulțiri

### d) C1 – costul unei adunări

### C2 – costul unei înmulțiri

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

$$C2/C1 > 3$$

$$a) S_{\text{Amdahl}} = 1,78 \text{ și } S_{\text{Gustafson}} = 4,5$$

### b) pentru Amdahl

dacă un program are timpul de execuție T atunci  $T = (1-p)T + pT$  (facem distincția între partea paralelizabilă și cea secvențială), dacă avem s core-uri atunci  $pT$  devine  $p/sT$ , deci accelerarea (raportul dintre timpul initial și nou timp cu s core-uri) este  $S = T/(1-(1-p)T + p/sT) = 1/(1-p + p/s)$

### pentru Gustafson

sistemul este capabil să execute  $E = (1-p)E + pE$  iar partea care se poate îmbunătății este doar  $pE$ , care devine  $\delta pE$ , deci execuția este îmbunătățită ( $(1-p)E + \delta pE)/E = 1 - p + \delta p$ )

$$c) S_{\text{Amdahl}} = 1 \text{ și } S_{\text{Gustafson}} = 1 \text{ (nicio îmbunătățire)}$$

$$d) S_{\text{Amdahl}} = 1 \text{ și } S_{\text{Gustafson}} = 1 \text{ (nicio îmbunătățire)}$$

$$e) L_{\text{Amdahl}} = 1/(1-p), S_{\text{Amdahl}} < 1/(1-p)$$

$$f) L_{\text{Gustafson}} = \infty$$

### g) verificați explicațile de la punct b) și țineți cont de rezultatele la limitele pentru p, s și δ

(d) calculați  $S_{\text{Amdahl}}$  și  $S_{\text{Gustafson}}$  dacă  $s = 1$  și  $\delta = 1$ ? Care este interpretarea rezultatului?

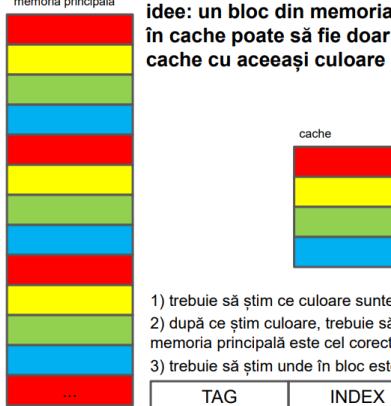
(e) calculați  $L_{\text{Amdahl}} = \lim_{s \rightarrow \infty} S_{\text{Amdahl}}$  și verificați dacă  $S_{\text{Amdahl}}$  este mereu sub sau peste această limită. Ce interpretează valoarea  $L_{\text{Amdahl}}$ ?

(f) calculați  $L_{\text{Gustafson}} = \lim_{\delta \rightarrow \infty} S_{\text{Gustafson}}$ ;

(g) explicați diferențele dintre legile lui Amdahl și Gustafson.

$$a) \frac{2^{32}}{2^5} = 2^{27}, (2^4 \times 2^{10}) / 2^5 = 2^9 = 512 \text{ blocuri}$$

- b) un exemplu în care cache are doar 4 blocuri, cea mai simplă idee: un bloc din memoria principală dacă e în cache poate să fie doar într-o poziție din cache cu aceeași culoare



tehnica aceasta de 1 la 1 se numește *direct mapping*  
dacă un bloc din memoria principală poate să fie în mai multe blocuri din cache (nu doar unul singur) atunci tehnica se numește *N-set associative mapping* (unde N este numărul de blocuri din cache în care un bloc din memorie poate fi copiat (este fie acolo, fie nu e în cache))  
această nouă tehnică oferă mai multă flexibilitate (1 la 1 este prea limitat)



- 1) trebuie să ştim ce culoare suntem
- 2) după ce ştim culoare, trebuie să ştim care block din memoria principală este cel corect (din toate cele roşii)
- 3) trebuie să ştim unde în bloc este byte-ul pe care îl vrem

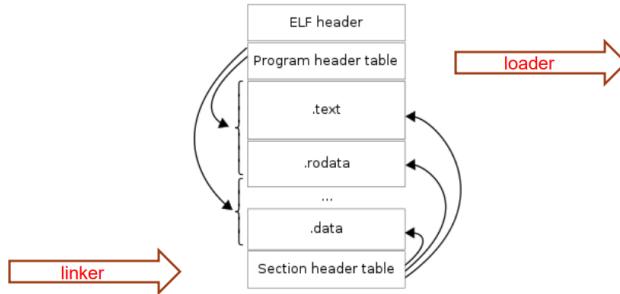
18 biți (ce rămâne pentru a identifica care bloc de aceeași culoare e cel corect)	9 biți (pentru că sunt $2^9 = 512$ blocuri/culoare în cache)	5 biți (pentru că sunt $2^5 = 32$ bytes posibili în bloc)
---	--	---

Considerăm un sistem de calcul pe 32 de biți care are un cache de 16 Kbytes. Cerințe:

- (a) considerăm că memorile sunt împărțite în blocuri de 32 de bytes. Câte blocuri sunt posibile în memoria principală? Câte blocuri sunt în memoria cache?
- (b) observați că numărul de blocuri din memoria principală poate să fie mult mai mare decât numărul de blocuri din memoria cache (în normal, memoria principală este mult mai multă decât memoria cache). Copiem bucati din memoria principală în memoria cache pe blocuri. Trebuie să ştim unde găsim în cache un byte pe care am vrea să îl citim din memoria principală. Sugerați moduri de a face corespondență între memoria principală și memoria cache.

## Curs 11,12 – Fișiere binare/Bootloader (Ew)

### • structura fișierelor ELF



**Linker:** pune pointer-i la secțiuni (sections) din executabil, nu e important la execuție  
**Loader:** pune pointer-i la segmentele (segments) din executabil, doar asta e folosit la execuție

### syscall pentru execuție

- EXEC

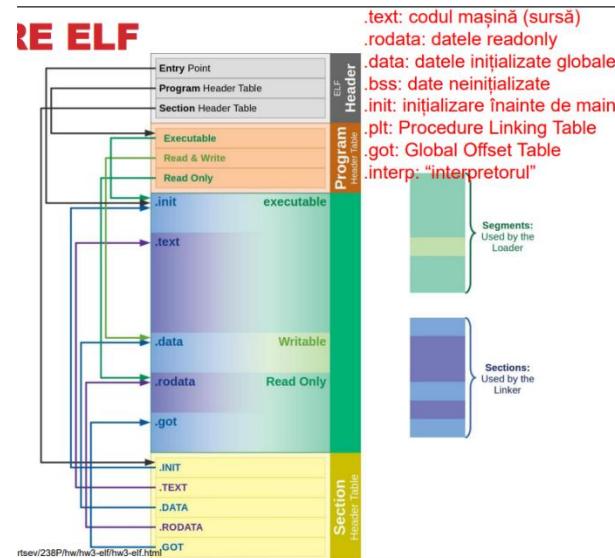
se citește header-ul fișierului

toate directivele LOAD sunt executate

execuția este preluată de entry point address (`_start` și apoi `main()`)

din aceste motive, avem mai mulți timpi afectați

- timpul de compilare (compile time)
  - o singură dată
  - codul este absolut (absolute code)
- timpul de încărcare (load time)
  - la fiecare execuție a binarului
  - codul este relocabil (relocatable code)
  - unele adrese sunt calculate la încărcarea binarului
- timpul de execuție (execute time)
  - este afectat de lazy binding



**simbolurile sunt referințe (către funcții sau variabile) în binar**

- nm a2.out
- în gdb când puneti „break main”, main aici este un simbol
  - denumirea funcției rămâne în binar dar nu este esențială la execuție
- puteți să scoateți simbolurile cu comanda „strip”
  - stripping symbols
  - debug și reverse engineering sunt mult mai dificile
  - fișierele binare sunt mai mici

### static linking

- simboluri din biblioteci externe sunt incluse în binar la link-are

### dynamic linking

- link-uri la simboluri din biblioteci externe sunt adăgiate la link-are iar la rulare loader-ul rezolvă aceste link-uri
- resolving symbols at runtime

când se calculează adresele simbolurilor? binding

- când programul este executat immediate binding
- când simbolul este folosit pentru prima dată lazy binding

staticce

- biblioteca e adăugată la compilare

dinamice/comune

- biblioteca este link-ată în timp real la execuție
- nu necesită recompilare
- este în regiunea comună de memorie (*shared memory*)
- *Position Independent Code (Position Independent Execution)*
  - *Global Offset Table*

- **cum calculăm acest padding?**

- notăm cu *start* adresa de start
- notăm cu *align* alinierea pe care o vrem (o putere a lui 2)

- $$\text{padding} = (\text{align} - (\text{start} \bmod \text{align})) \bmod \text{align}$$

$$= (\text{align} - (\text{start} \& (\text{align} - 1))) \& (\text{align} - 1)$$

$$= \text{start} \& (\text{align} - 1)$$

- $\text{aliniat} = \text{start} + \text{padding}$

$$= \text{start} + ((\text{align} - (\text{start} \bmod \text{align})) \bmod \text{align})$$

$$= (\text{start} + (\text{align} - 1)) \& \sim(\text{align} - 1)$$

$$= (\text{start} + (\text{align} - 1)) \& \sim\text{align}$$

la pornirea calculatorului este activat BIOS-ul

**BIOS-ul este în RAM:**

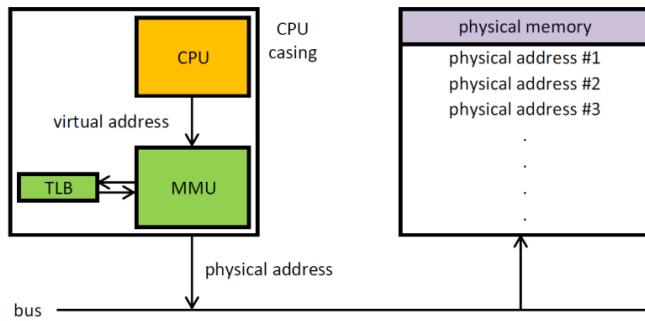
- realizează Power-On Self Test (POST procedure)
- încarcă bootloader-ul
- scopul este găsirea sistemului de operare și rularea sa
- OS-ul este căutat pe HDD/SSD/CD-ROM/USB/floppy

**unde este bootloader-ul?**

- primul sector (primii 512 bytes) de pe dispozitiv
- de unde știm că e bootloader? magic number: 0xAA55

**bootloader-ul găsit este încărcat în memorie la 0x7C00**

- **ce se întâmplă în hardware**



CPU: Central Processing Unit

MMU: Memory Management Unit

TLB: Translation lookaside buffer

TLB e un cache ca această traducere din adrese logice în fizice să se facă rapid

pentru că "primul bootloader" este limitat la 512 bytes, acesta încarcă defapt încă un bootloader care nu mai are limitări

pe Windows, bootloader-ul este la  
Windows\System32\ntoskrnl.exe

în tot acest timp, procesorul este în modul de lucru pe 16 biți

**Quick Emulator (QEMU)**

**emulator open-source**

- emulează un procesor (și periferice etc.)
- folosește traducere binară dinamică (dynamic binary translation)
- putem testa secvența de boot