

Curs Arhitectura Sistemelor de Calcul - FMI

Matei-Iulian Cocu
matei-iulian.cocu@s.unibuc.ro

Stefan Chiper
stefan.chiper@s.unibuc.ro

Cuprins

1	Prolog	3
2	Evoluția sistemelor de calcul. Sistemul binar	3
2.1	Scurt istoric al sistemelor de calcul	3
2.2	Sistemul binar - baza sistemelor moderne de calcul	4
2.3	Lectură suplimentară	5
3	Teoria Informației	5
3.1	Măsurarea cantității de informație	5
3.2	Codarea Datelor - Entropia lui Shannon	5
3.3	Detectarea/Corectarea Erorilor	5
3.4	Lectură suplimentară	7
4		7
4.1	7
4.2	7
4.3	7
4.4	Lectură suplimentară	7
5		7
5.1	7
5.2	7
5.3	7
5.4	Lectură suplimentară	7
6		7
6.1	7
6.2	7
6.3	7
6.4	Lectură suplimentară	7
7		7
7.1	7
7.2	7
7.3	7
7.4	Lectură suplimentară	7
8		7
8.1	7
8.2	7
8.3	7
8.4	Lectură suplimentară	7
9		7
9.1	7
9.2	7
9.3	7
9.4	Lectură suplimentară	7

10 The Eight Great Ideas in Computer Architecture

10.1 Design for Moore's Law	7
10.2 Use Abstraction to Simplify Design	8
10.3 Make the Common Case Fast	8
10.4 Performance via Parallelism	8
10.5 Performance via Pipelining	9
10.6 Performance via Prediction	9
10.7 Hierarchy of Memories	10
10.8 Dependability via Redundancy	10

1 Prolog

Acest document este o compilație succintă a cursului de **Arhitectura Sistemelor de Calcul**, o materie de bază a primului an în cadrul specializării Informatică din cadrul Facultății de Matematică și Informatică București. Documentul conține toată materia imperativă cursului, cât și alte adăugiri relevante materiei de studiu.

Fiind un curs introductiv, acesta introduce noțiuni importante în următoarele domenii:

- securitate informatică
 - Reverse Engineering (RE)
 - hacking
- optimizare
 - dezvoltare a jocurilor
 - Machine Learning (ML/AI)
- debugging
- dezvoltare software *low-level*
 - dezvoltare pentru sisteme embedded
 - dezvoltare pentru sisteme de operare

Referințe bibliografice generale

- D. Patterson and J. Hennessy, Computer Organisation and Design
- R. Blum, Professional Assembly Language

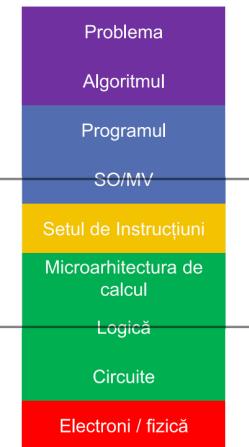


Figure 1: Ierarhia Abstractizării

2 Evoluția sistemelor de calcul. Sistemul binar

2.1 Scurt istoric al sistemelor de calcul

2.1.1 Blaise Pascal (1623 - 1662)

În 1642, crează **Pascaline**: un calculator mecanic capabil de adunări și scăderi (utilizat pentru calcul de taxe). Nefiind o mașină practică, mai puțin de 50 de astfel de mașini au fost create, fiind utilizată mai degrabă pe post de "jucărie" a aristocraților. Bineînteles, acesta a fost un important contribuitor al secolului XVII, limbajul **Pascal** fiind numit în onoarea acestuia.

peste 13 tone

2.1.5 Ada Lovelace (1815 - 1852)

- colaboratoare a lui *Babbage*
- scrie primul program, calculează *numere Bernoulli*
- nu existau limbaje de programare, programul reprezentând *o serie de pași de executat de o mașină*
- este considerată primul "programator"

2.1.6 Konrad Zuse (1910 - 1995)

- introduce o serie de calculatoare: Z1, Z2, Z3 și Z4
- primele prototipuri în 1940-1941, folosesc *sistemul binar*, instrucțiunile acestora fiind stocate pe o *bandă perforată*
- introduce reprezentarea și calculul în *virgulă mobilă*
- majoritatea efortului acestuia a fost făcut în izolare (1936-1945), dat fiind contextul cel de-al Doilea Război Mondial

2.1.7 Alan Turing (1912 - 1954)

- celebru pentru publicul larg pentru contribuția lui în spargerea rapidă a mesajelor **Enigma** utilizând mașina "The Bombe" (calcul brut cu scopul de a reduce numărul de posibilități în decriptarea mesajelor)
- introduce Mașina Turing
 - un model teoretic pentru a implementa orice algoritm
 - conceptul de Turing-complete (un sistem care poate recunoaște și analiza seturi de reguli pentru manipularea datelor - o cantitate infinită, teoretic)

2.1.3 George Boole (1815 - 1864)

- scrie *"The Laws of Thought"* (1854)
- introduce *logica booleană* și analizează operațiile de bază (*negația (NOT)*, *conjunctia (AND)*, *disjunctia (OR)*, *disjunctia exclusivă (XOR)*)

2.1.4 Charles Babbage (1791 - 1871)

- proiectează teoretic Mașina Diferențială (*Difference Engine No. 2*)
- design-ul este realizat de abia în 1991
- reprezintă prima mașină de calcul (*mecanică*) programabilă
- asemenea prototipuri ajungea la greutatea de gradul tonelor, această mașină ajungând să cântărească

2.1.8 John von Neumann (1903 - 1957)

Considerat unul dintre cei mai influenți matematicieni ai ultimului secol, John von Neumann a adus contribuții fundamentale în numeroase domenii. În domeniul calculatoarelor, cele mai importante sunt:

- contribuția la crearea primului calculator electronic ENIAC (Electronic Numerical Integrator And Computer), 1939-1944
- îmbunătățirea ENIAC, ajutând la crearea EDVAC (Electronic Discrete Variable Automatic Computer), sistemul este binar și are programe stocate
- introducerea arhitecturii von Neumann - diferența dintre aceasta și arhitectura Harvard fiind utilitatea în sine:
 - arhitectura Harvard prioritizează viteza de procesare, având la dispoziție două seturi de memorie separate;
 - arhitectura von Neumann conferă sistemului de calcul o flexibilitate superioară arhitecturii latente;
 - uzual, arhitectura von Neumann este folosită în sistemele de calcul *general purpose* (desktop-uri, laptop-uri), pe când arhitectura Harvard este folosită de obicei în scopuri specifice (ex.: mașini de spălat, sisteme anti-furt, tehnologie militară)

2.1.9 Claude Shannon (1916 - 2001)

- considerat "părintele teoriei informației"
- contribuții excepționale atribuite
 - demonstrează faptul că problemele de logica Booleană pot fi rezolvate cu circuite electronice
 - teorema de esantionare **Shannon-Nyquist** (de la analog la digital și înapoi, fără pierderi de informație)
 - inventează teoria informației

2.1.10 Post-Shannon

- după cel de-al Doilea Război Mondial, cercetarea în domeniul calculatoarelor începe în ritm exponentional
- actorii importanți în domeniu au devenit grupurile profesionale (ex.: IEEE, ACM, Bell Labs) și statele (ex.: Statele Unite, Germania, Marea Britanie, programele de cercetare DARPA)

2.1.11 Alte personalități (și concepte) importante

- **Al-Khwarizmi (780 - 845)**: astronom/astrolog persan, matematician, inventatorul termenului "algoritmul";
- **Grace Hopper (1906 - 1992)**: creatorul primului compilator, dezvoltator al limbajului COBOL;
- **Margaret Hamilton (1936 -)**: software engineer, a condus echipa care a dezvoltat software-ul de zbor pentru misiunile Apollo;

2.2 Sistemul binar - baza sistemelor moderne de calcul

- bit = *b*inary *d*igit
- sistem de numărare cu *baza B = 2*
- avem disponibile două cifre: *0* și *1*

$x = \sum_{i=0}^{N-1} b_i 2^i$, unde *N* este numărul de biți folosiți în

reprezentare

bit b_i :	0	1	1	1	1	0	0	0	1
2^i :	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

În exemplul de mai sus: $0 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 241$ (avem

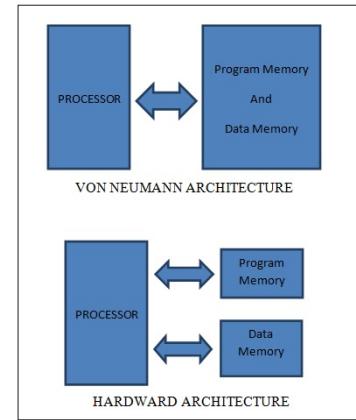


Figure 2: Ierarhia Abstractizării

- **Barbara Liskov (1939 -)**: distributed computing, limbajul CLU, principiul substituției Liskov, premiul Turing 2008, prima femeie doctorandă în CS/IT;
- **Howard H. Aiken (1900 - 1973)**: ASCC, Mark I, Mark II, Mark III, Mark IV;
- **Richard Hamming (1915 - 1998)**: coduri de corecție a erorilor, distanța Hamming, premiul Turing 1968;
- **RSA (1977)**: Ron Rivest, Adi Shamir, Leonard Adleman - criptografie cu cheie publică;
- **Diffie-Hellman (1976)**: Whitfield Diffie, Martin Hellman, Ralph Merkle - metodă matematică de generare securizată a unei chei criptografice simetrice;
- **UNIX - Ken Thompson, Dennis Ritchie**: sistem de operare inițiat de un grup de programatori de la Bell Labs în 1969, scris în limbajul C, influențând numeroase sisteme de operare ulterioare (inclusiv Linux și MacOS);
- **Linus Torvalds**: creatorul nucleului Linux în 1991, inițial ca un proiect personal, devenind ulterior unul dintre cele mai importante sisteme de operare open-source;
- **Richard Stallman**: fondatorul mișcării software liber, inițiatorul proiectului GNU în 1983, care a contribuit semnificativ la dezvoltarea software-ului open-source;
- **Brian Kernighan și Dennis Ritchie**: co-creatori ai limbajului de programare C;
- **Larry Page (CEO) și Sergey Brin (CTO)**: Google, Alphabet Incorporated.

$N = 9$, dar în realitate se poate reduce la $N = 8$)

Intuiția noastră este în baza $B = 10$, dar este folositor să abstractizăm și să considerăm baza generală B ; aşadar, în baza B avem:

- cifre de la *0* la *B-1* (restul se numesc numere)
- reprezentarea este $x = \sum_{i=0}^{N-1} b_i B^i$
- bitul b_0 se numește *Least Significant Bit (LSB)* iar bitul b_{N-1} se numește *Most Significant Bit (MSB)*
- reprezentarea unui număr din baza 10 în baza B se

face prin împărțiri succesive cu B și păstrare de rest
Regula generală: când trecem din baza B în baza B^p trebuie doar să grupăm noul număr în câte p cifre

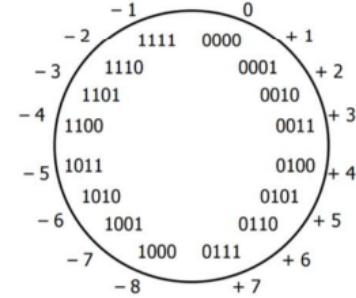
Numere întregi negative	bit b_i : <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>2^7:</td><td>-2^7</td><td>2^6</td><td>2^5</td><td>2^4</td><td>2^3</td><td>2^2</td><td>2^1</td><td>2^0</td></tr> </table>	1	1	1	1	0	0	0	1	2^7 :	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	1	1	0	0	0	1											
2^7 :	-2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0										

Reprezentarea în complement față de 2:

- numere în intervalul $[-2^{N-1}, 2^{N-1} - 1]$
- se pierde un bit pentru semn, dar este optim: **MSB** este semnul, **restul bițiilor** reprezintă valoarea
- pentru a obține reprezentarea în complement față de 2 a unui număr negativ, se inversează toți biții și se adaugă 1
- motivul pentru care folosim acest sistem în complement față de 2 este că *algoritmul de adunare* este la fel ca pentru numere naturale

Adunări/Scăderi pe numere naturale și întregi:

- se adună bit cu bit, de la LSB la MSB; se păstrează un **carry** dacă suma depășește baza (2)
- **Overflow** apare când rezultatul nu poate fi reprezentat cu numărul de biți alocati
- **Underflow** apare când rezultatul este mai mic decât cel mai mic număr reprezentabil



Extinderea numărului de biți:

- **Zero-Extension** - pentru numere naturale, se adaugă biți 0 în partea stângă (e.g., 6 biți \rightarrow 8 biți: 001101 \rightarrow 00001101)
- **Sign-Extension** - pentru numere întregi, se adaugă biți egali cu MSB în partea stângă (e.g., 6 biți \rightarrow 8 biți: 111101 \rightarrow 11111101)

Logica binară (unde 0 = False, 1 = True) implică operații de bază esențiale în arhitectura sistemelor de calcul pentru funcționarea circuitelor digitale și a procesorului: NOT (*negatie*), AND (*conjunctie*), OR (*disjunctie*) și XOR (*disjunctie exclusivă*).

NOT	
A	$\neg A$
0	1
1	0

AND		
A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

OR		
A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

2.3 Lectură suplimentară

- **PH Book:**
 - [2.4: Signed and Unsigned Numbers](#)
 - [2.17: Real Stuff: x86 Instructions](#)
 - [3.2: Addition and Subtraction](#)
- [Thomas Finley Course Notes](#)
- **Mașina Turing** și conceptul de **Turing-complete**:
 - [Turing Machines Explained](#)
 - [Turing-complete](#)
- [Computer Pioneers - Part 1](#)
- [Computer Pioneers - Part 2](#)
- [BBC History of Computers](#)
- [The Grand Narrative of the History of Computing](#)

3 Teoria Informației

3.1 Măsurarea cantității de informație

3.2 Codarea Datelor - Entropia lui Shannon

3.3 Detectarea/Corectarea Erorilor

$$x = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

În exemplul de mai sus: $-1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = -15$

3.4 Lectură suplimentară

4

4.1

4.2

4.3

4.4 Lectură suplimentară

5

5.1

5.2

5.3

5.4 Lectură suplimentară

6

6.1

6.2

6.3

6.4 Lectură suplimentară

7

7.1

7.2

7.3

7.4 Lectură suplimentară

8

8.1

8.2

8.3

8.4 Lectură suplimentară

9

9.1

9.2

9.3

9.4 Lectură suplimentară

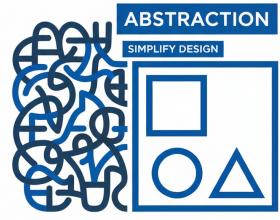
10 The Eight Great Ideas in Computer Architecture

10.1 Design for Moore's Law

MOORE'S LAW



10.2 Use Abstraction to Simplify Design



Both computer architects and programmers had to invent techniques to make themselves more productive, for otherwise design time would lengthen as dramatically as resources grew by *Moore's Law*. A major productivity technique for hardware and software is to use **abstractions** to represent the design at different levels of representation; lower-level details are hidden to offer a simpler model at higher levels. We'll use the abstract painting icon to represent this second great idea.

10.3 Make the Common Case Fast

MAKE THE
COMMON CASE FAST
OPTIMIZE FOR SPEED



Making the **common case fast** will tend to enhance performance better than optimizing the rare case. Ironically, the common case is often simpler than the rare case and hence is often easier to enhance. This common sense advice implies that you know what the common case is, which is only possible with careful experimentation and measurement. We use a sports car as the icon for making the common case fast, as the most common trip has one or two passengers, and it's surely easier to make a fast sports car than a fast minivan.

10.4 Performance via Parallelism

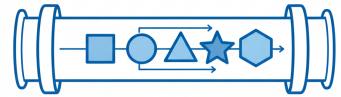
PERFORMANCE VIA
PARALLELISM
MULTIPLE CORES, SIMD



Since the dawn of computing, computer architects have offered designs that get more performance by performing operations in parallel. We use the multiple jet engines of a plane as our icon for **parallel performance**.

10.5 Performance via Pipelining

PERFORMANCE VIA PIPELINING
INSTRUCTION STAGES



A particular pattern of parallelism is so prevalent in computer architecture that it merits its own name: **pipelining**. For example, before fire engines, a "bucket brigade" would respond to a fire, which many cowboy movies show in response to a dastardly act by the villain. The townsfolk form a human chain to carry a water source to fire, as they could much more quickly move buckets up the chain instead of individuals running back and forth. Our pipeline icon is a "sequence" of pipes, with each section representing one stage of the pipeline.

10.6 Performance via Prediction

PERFORMANCE VIA
PREDICTION
BRANCH PREDICTION, CACHE MISS

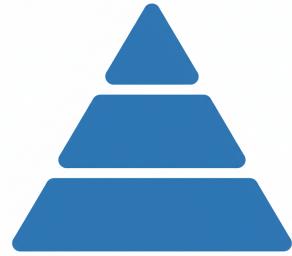


Following the saying that it can be better to ask for forgiveness than to ask for permission, the final great idea of performance is **prediction**. In some cases it can be faster on average to guess and start working rather than wait until you know for sure, assuming that the mechanism to recover from a misprediction is not too expensive and your prediction is relatively accurate. We use the fortune-teller's crystal ball as our prediction icon.

10.7 Hierarchy of Memories

HIERARCHY OF MEMORY

SPEED & COST



Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy of memories**, with the fastest, smallest, and most expensive memory per bit at the top of the hierarchy and the slowest, largest, and cheapest per bit at the bottom. Caches give the programmer the illusion that main memory is nearly as fast as the top of the hierarchy and nearly as big and cheap as the bottom of the hierarchy. We use a layered triangle icon to represent the memory hierarchy. The shape indicates speed, cost, and size: the closer to the top, the faster and more expensive per bit the memory; the wider the base of the layer, the bigger the memory.

10.8 Dependability via Redundancy

DEPENDABILITY VIA REDUNDANCY

BACKUP SYSTEMS, FAULT TOLERANCE



Computers not only need to be fast; they need to be dependable. Since any physical device can fail, we make systems **dependable** by including redundant components that can take over when a failure occurs *and* to help detect failures. We use the tractor-trailer as our icon, since the dual tires on each side of its rear axles allow the truck to continue driving even when one tire fails.