Design for Moore's Law

The one constant for computer designers is rapid change, which is driven largely by **Moore's Law**. It states that integrated circuit resources double every 18–24 months. Moore's Law resulted from a 1965 prediction of such growth in IC capacity made by Gordon Moore, one of the founders of Intel. As computer designs can take years, the resources available per chip can easily double or quadruple between the start and finish of the project. Like a skeet shooter, computer architects must anticipate where the technology will be when the design finishes rather than design for where it starts. We use an "up and to the right" Moore's Law graph to represent designing for rapid change.


MOORE'S LAW

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.


ABSTRACTION

Make the Common Case Fast

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 (see Section 1.6). 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!


COMMON CASE FAST


PARALLELISM

Performance via Parallelism

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


PIPELINING

Performance via Pipelining

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.


PREDICTION

Performance via Prediction

Following the saying that it can be better to ask for forgiveness than to ask for permission, the final great idea 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.


HIERARCHY

Hierarchy of Memories

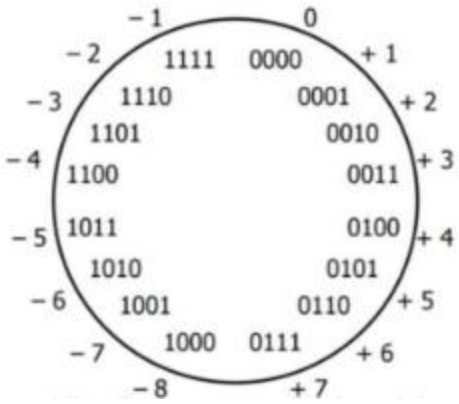
Programmers want memory to be fast, large, and cheap, as memory speed often shapes performance, capacity limits the size of problems that can be solved, and the cost of memory today is often the majority of computer cost. Architects have found that they can address these conflicting demands with a **hierarchy 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. As we shall see in Chapter 5, 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.


DEPENDABILITY

Dependability via Redundancy

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 axels allow the truck to continue driving even when one tire fails. (Presumably, the truck driver heads immediately to a repair facility so the flat tire can be fixed, thereby restoring redundancy!)

- Curs 0x01



- **operații aritmetice, numere întregi exemple**
  - ambele numere pozitive, mari

	0	1	1	1	0	1	1	1	119
	0	1	0	0	1	0	1	1	75
+	1	1	0	0	0	0	1	0	-62

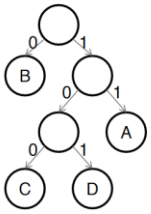
- **intuiția, de unde am obținut -62?**
  - 119 + 75 = 194 (nu încape pe 7 biți)
  - maximum e 127, deci avem 194 - 127 = 67 “extra”
  - overflow începe după 127, după 127 este -128 (folosim un extra)
  - deci 66 extra rămași pornesc de la -128
  - deci avem -128 + 66 = -62

- Curs 0x02

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

H(X) se numește entropia lui X
I(X) este informația despre X
E este “expected value”, operația care calculează valoarea medie

- **cum putem crea o codare eficientă și unică?**
  - un arbore binar
    - frunzele sunt codurile
    - stânga/dreapta e decis de 0/1
    - codarea este:
      - B = 0
      - A = 11
      - C = 100
      - D = 101
    - asta garantează codare eficientă și decodare unică

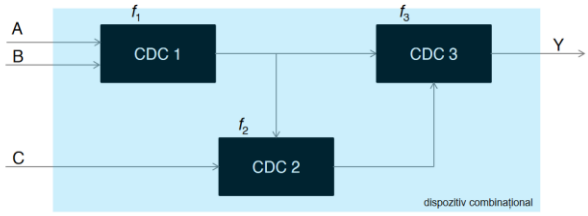


  - cum generăm codarea eficientă?
    - algoritmul Huffman
    - input: probabilitatea fiecărui eveniment {1/3, ½, 1/12, 1/12}
    - output: codurile care se citesc de pe un arbore binar (mai sus)
    - cheia: unele evenimente/simboluri apar mai des decât altele, deci acestea primesc o codare mai scurtă
    - dacă toate evenimente sunt equiprobabile, atunci nu putem face nimic

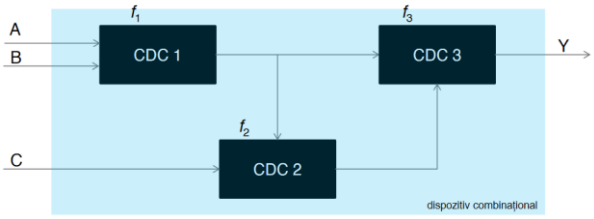
distanța Hamming între două șiruri binare: câți biți sunt diferiți (biți de pe aceleași poziții în preprezentarea binară)

o distanță Hamming de *2E+1* poate corecta *E* erori
- care e intuiția? majority vote

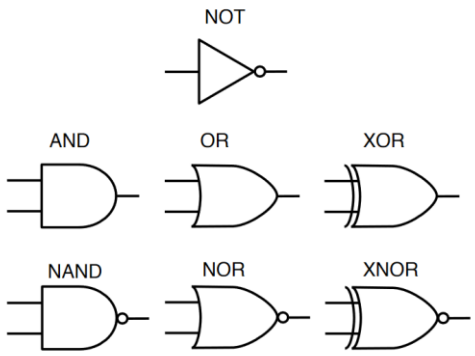
- Curs 0x03



- care este funcția dispozitivului? Y = $f_3(f_1(A, B), f_2(f_1(A, B), C))$



- timpul total de propagare? $t_{p,total} = t_{p,1} + t_{p,2} + t_{p,3}$ (longest path)



- **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	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?

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

- **concluzia: NOT, AND și OR sunt universale (pot implementa orice circuit combinațional)**
- **de ce lipsește XOR? $A \oplus B = \overline{A}B + A\overline{B}$**

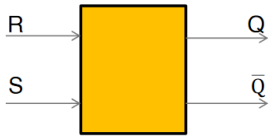
A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

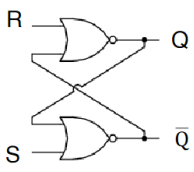
- **care e numărul maxim de termeni în sumele din X? ½2^N**

- Curs 0x04

SR Latch (Set-Reset Latch)

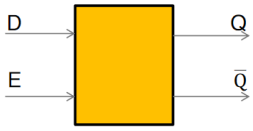
- memorează un bit de informație

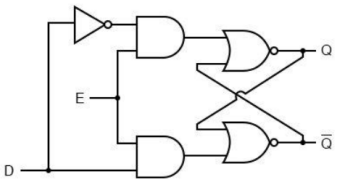




S	R	Q	$\overline{Q}$	
0	0	latch	latch	nu se schimbă nimic
0	1	0	1	aici punem "0" în memorie
1	0	1	0	aici punem "1" în memorie
1	1	0	0	stare invalidă

- **SR Latch (Set-Reset Latch)**
  - e bun dar are două intrări, putem face ceva cu o singură intrare?
  - **D Latch**

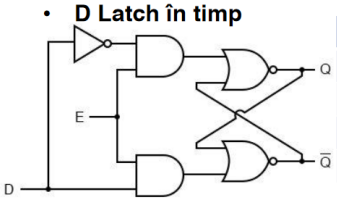




E	D	Q	$\overline{Q}$	
0	0	latch	latch	nu se schimbă nimic
0	1	latch	latch	nu se schimbă nimic
1	0	0	1	aici punem "0" în memorie
1	1	1	0	aici punem "1" în memorie

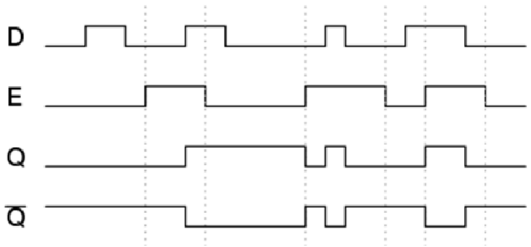
E de la Enable, adică activare
dacă E = 0 nu se întâmplă nimic

- **D Latch în timp**

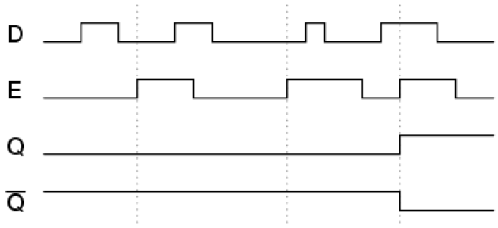


E	D	Q	$\overline{Q}$	
0	0	latch	latch	nu se schimbă nimic
0	1	latch	latch	nu se schimbă nimic
1	0	0	1	aici punem "0" în memorie
1	1	1	0	aici punem "1" în memorie

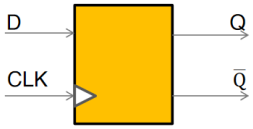
E de la Enable, adică activare
dacă E = 0 nu se întâmplă nimic



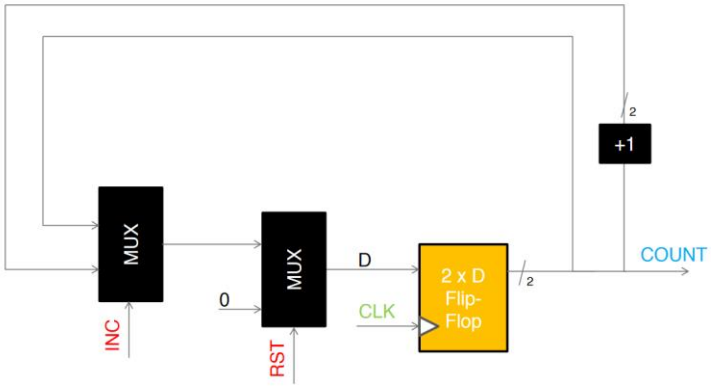
- **D Flip-Flop**



  - E devine clock – adică ceasul sistemului
    - la un interval fix de timp (un ciclu), sistemul face ceva



un set de câteva D Flip Flops care au același CLK = un registru



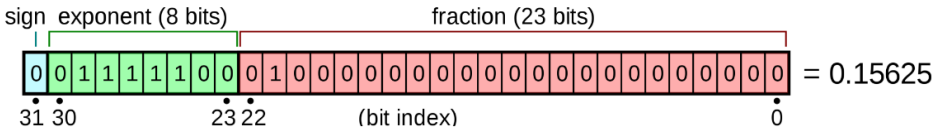
este un counter pe 2 biți

starea anterioară	starea viitoare		
	inc = 0, rst = 0	inc = 1, rst = 0	rst = 1
00	00	01	00
01	01	10	00
10	10	11	00
11	11	00	00

- Curs 0x05

REPREZENTAREA ÎN VIRGULĂ MOBILĂ

- **standardul: IEEE 754 Floating Point**

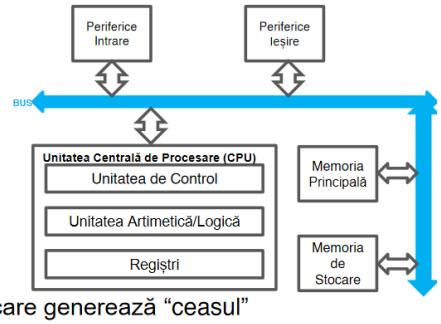


- **$x = (-1)^s \ 1.mmmmmmmmmmmmmmmmmmmmmmm \ 2^{(eeeeeeee)_2 - 127}$**

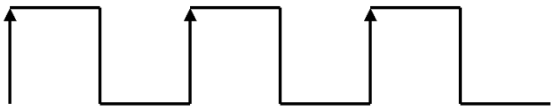
- **exemple:**
  - $0.15625 = (-1)^0 \ 1.0100...0 \ 2^{b01111100 - 127} = 1.25 \ 2^{-3} = 1.25/8$
  - alte exemple:
  - $(-1)^0 \ 1.1000...0 \ 2^{b01111100 - 127} = 1.5 \ 2^{-3} = 1.5/8 = 0.1875$
  - $1 = (-1)^0 \ 1.00000000000000000000000 \ 2^{b01111111 - 127}$
  - $-1 = (-1)^1 \ 1.00000000000000000000000 \ 2^{b01111111 - 127}$
  - $2 = (-1)^0 \ 1.00000000000000000000000 \ 2^{b10000000 - 127}$
  - $\infty = (-1)^0 \ 1.00000000000000000000000 \ 2^{b11111111 - 127}$
  - $-\infty = (-1)^1 \ 1.00000000000000000000000 \ 2^{b11111111 - 127}$

- Curs 0x06

- **Unitatea Centrală de Procesare**
  - a.k.a. CPU
  - este “creierul” unității de calcul
  - execută instrucțiuni

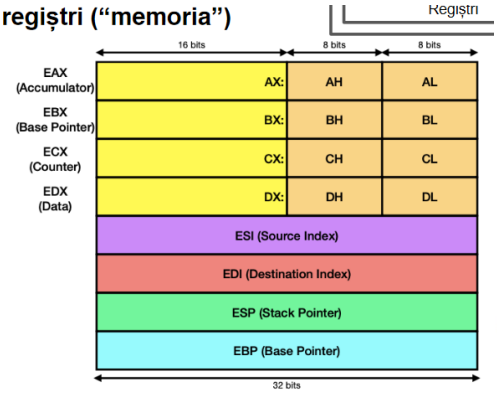


  - 5 componente principale:
    - **Clock**
      - este un circuit special care generează “ceasul”

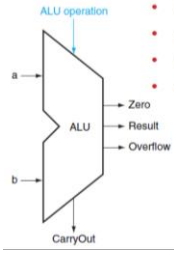


      - este frecvența la care operează (calcule și sincronizarea componentelor secvențiale) CPU-ul
      - cu cât este mai mare frecvența, cu atât mai bine (în general)
      - se măsoară în MHz sau GHz

- **regiștri (“memoria”)**



- **UAL (“operații”)**
  - operații aritmetice cu întregi
  - operații logice
  - operații aritmetice cu numere în formatul floating point
  - operații speciale: sqrt, exp, trig



după orice operație, știm “gratuit” dacă rezultatul a fost sau nu zero – este folositor?

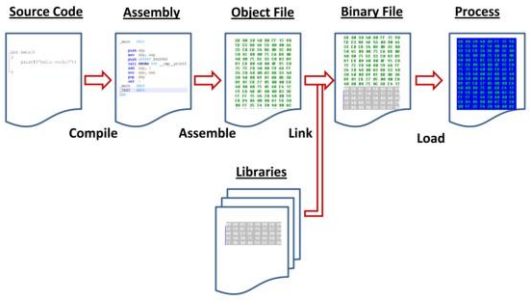
- **BUS**
  - CPU are nevoie de șiruri de biți din memoria principală sau cea de stocare
  - CPU are nevoie să scrie înapoi în memorie rezultate
  - CPU coordonează perifericele

- **UC (“instrucțiunile”)**
  - fetch
    - citim din memorie codul care trebuie executat
    - de unde din memorie? Instruction Pointer ne spune
  - decode
    - circuitul “Instruction Decoder” analizează biții citiți din memorie ca să “înțeleagă” ce să facă cu ei
  - execute
    - execută instrucțiunea decodată
    - poate duce la schimbarea IP sau la transmiterea ceva pe BUS către memorie
  - calculează următorul IP

IF – Instruction Fetch (citirea din memorie a instrucțiunilor)
ID – Instruction Decode (circuit secvențial care decodează)
EX – Execute (execuția propriu-zisă)
MEM – Memory Access (orice access memorie)
WB – Write Back (scrie rezultatul înapoi în memorie)

  - fetch
    - IP = 10011 (locația în memorie de unde să citim biții)
    - după citire, IP este actualizat
  - decode
    - s-a citit “11000110” care este decodat în
      - opcode = 110, operand1 = 00 operand2 = 110
      - de exemplu: 110 = “adună valoarea imediată A la registrul R”, R = 00 este EAX (prin convenție), A = 110 (adică 6)
  - execute
    - trimite EAX ← EAX + 6 la UAL
    - citește rezultatul din UAL și pune-l în registrul EAX

- Curs 0x07



C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

Instruction Set Arhitecture (ISA)

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

- **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

- Curs 0x08

PIPELINING

- **imaginea arată bine, din păcate nu putem face așa ceva mereu**
  - pipeline stalls (întârzieri în conducta de date)
  - aceste evenimente se numesc hazards (erori)
    - structural hazards: o unitate de calcul este deja utilizată
      - două instrucțiuni încearcă să acceseze aceeași unitate
    - data hazards: datele nu sunt pregătite pentru utilizare
      - o instrucțiune depinde de rezultatul unei instrucțiuni precedente
    - control hazards: nu știm următoarea instrucțiune
      - din cauza unor instrucțiuni de jump nu știm instrucțiunea următoare

data hazards

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

- Curs 0x09

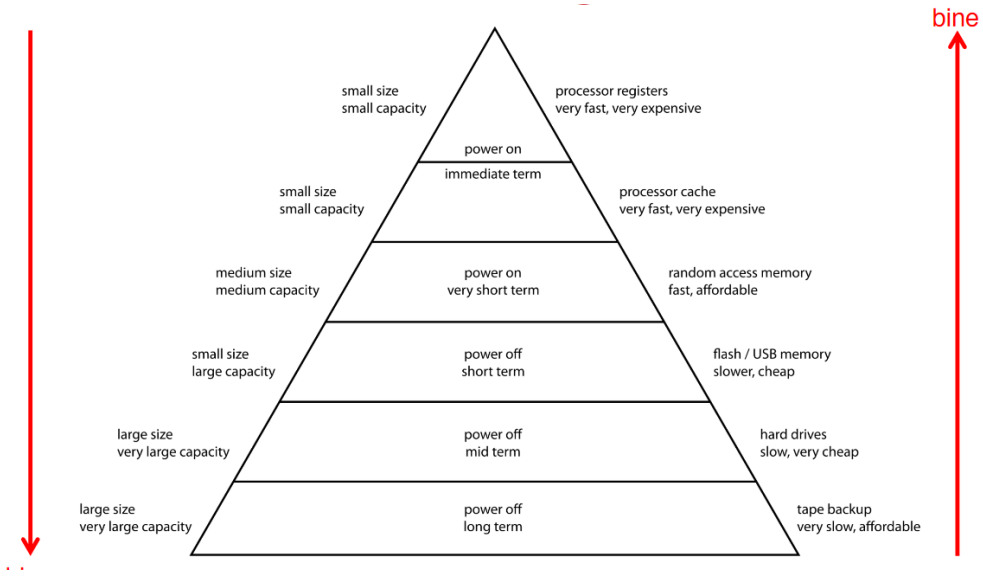
putem rula un program mai eficient (paralelizând părți ale sale)

- 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*):

$$S = 1 - p + \text{ș}p$$



- **de ce e bine să avem cache?**
- **presupunem că timpii de acces sunt:**
  - în memoria principală: 50 ns
  - în L1: 1 ns (dar există o probabilitate de 10% ca în L1 să nu găsim ceea ce căutăm, i.e., 10% miss rate)
  - în L2: 5 ns cu 1% miss rate
  - în L3: 10 ns cu 0.2% miss rate
- **să presupunem că vrem să accesăm o bucată de memorie, cât ne costă ca timp dacă:**
  - verificăm în RAM: 50 ns
  - verificăm în L1: 1 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 legatură miss rate? cu dimensiunea memoriei

- Curs 0x0A

rulăm un program A pe un sistem de calcul X

- performanța$_X$ = (timpul de execuție a lui A pe X)$^{-1}$
- 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$_X$ (performanța$_Y$)$^{-1}$= *n*
- timpul de execuție îl măsurăm în secunde
  - se numește *wall-clock time*, *response time* sau *elapsed time*
  - este timpul în "lumea reală"
  - de ce e complicat? avem mai mulți timpi?
    - un procesor execută simultan mai multe programe
    - *CPU time* = cât timp de execuție a fost alocat pe CPU

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

CISC VS. RISC

- **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-registru
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.)

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

Complex Instruction Set Computers

- **MUL** MEM_LOC_1, MEM_LOC_2

Reduced Instruction Set Computers

- **LOAD** MEM_LOC1, R1
- **LOAD** MEM_LOC2, R2
- **MUL** R1, R2
- **STORE** R2, MEM_LOC2