

Arhitectura calculatoarelor

AN II CALCULATOARE
Seria B

E-mail: florin.oniga@cs.utcluj.ro

Web: <http://users.utcluj.ro/~onigaf>, secțiunea Teaching/AC

Obiectivele cursului

Pregătirea studenților pentru:

- Aplicarea metodelor de reprezentare și de proiectare la nivel de sistem
- Specificarea Arhitecturii Setului de Instrucțiuni
- A scrie programe simple în limbaje de asamblare și cod mașina
- A specifica, proiecta, implementa și testa Unități Centrale de Prelucrare (UCP) -
Micro-arhitecturi - Căi de date și Unități de control
- A înțelege organizarea Memoriei și I/O
- A înțelege tendințele moderne în arhitectura calculatoarelor

Conținutul cursului

Introducere

Proiectarea Sistemelor Digitale

Arhitectura Setului de Instrucțiuni

Proiectarea procesorului cu un singur ciclu de execuție

Aritmetica calculatoarelor, unități aritmetice și logice

Procesorul cu cicluri multiple de execuție – calea de date

Procesorul cu cicluri multiple de execuție – controlul

Procesorul pipeline

Procesorul pipeline avansat – planificarea statică și dinamică a execuției

Predicția ramificărilor

Procesoare superscalare

Memorii

Arhitecturi moderne de procesoare

Rezolvarea de probleme

Cursul se va actualiza periodic pe:

Echipa Teams/Files

Obiectivele lucrărilor de Laborator

Lucrările de laborator și temele acasă sunt componente obligatorii ale disciplinei Arhitectura Calculatoarelor. Obiectivele lucrărilor de laborator + teme acasă:

- A învăța studenții să opereze cu conceptele și metodele prezentate în curs.
- A dezvolta abilități practice pentru programare în cod mașina, specificarea și proiectarea micro - arhitecturilor folosind RTL, implementări de micro-arhitecturi folosind VHDL.

Temele principale de laborator:

- Proiectare cu instrumente Xilinx Vivado Webpack și plăci de dezvoltare cu FPGA.
- Componente hardware sintetizabile în VHDL; implementate și testate pe plăci cu FPGA.
- Limbaj de asamblare MIPS (set restrâns), rularea unor programe simple pe CPU implementat.
- Proiectarea (VHDL), implementarea și testarea (placi cu FPGA) a unor micro-arhitecturi MIPS.

Laboratorul se va actualiza periodic pe:

<http://users.utcluj.ro/~onigaf/> secțiunea Teaching/AC

Conținutul Laboratorului

Introducere Vivado Xilinx

Componente combinaționale

Componente secvențiale

Procesor 1 – MIPS ciclu unic – 5 Laboratoare

Procesor 2 – MIPS pipeline – 2 Laboratoare

Interfatare UART – 2 Laboratoare

Evaluare

- Examen scris (E).
- Evaluarea activității la lucrările de laborator, temelor de casa, teste (LA).

$$\text{Nota} = 0.5 * E + 0.5 * LA$$

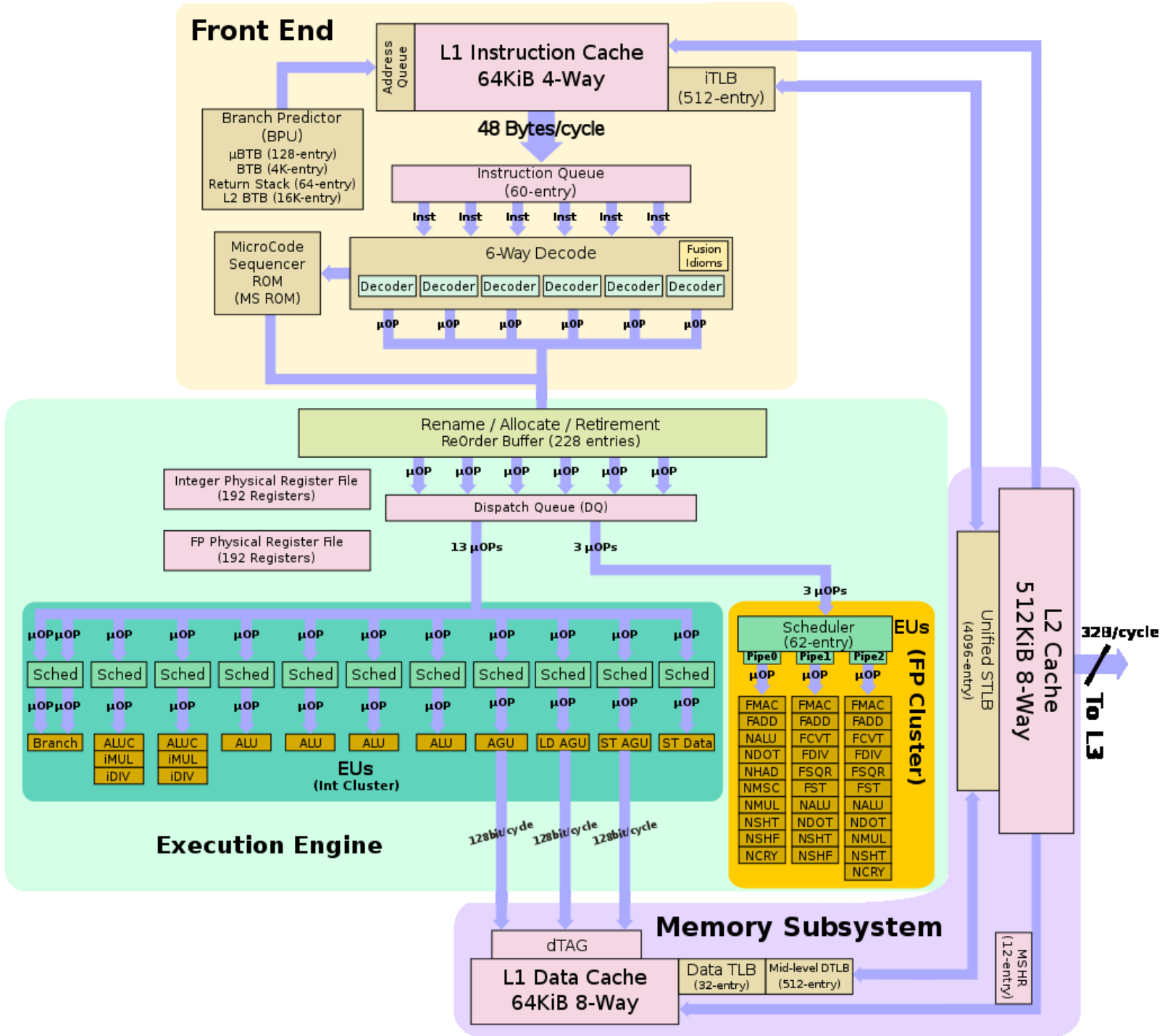
Condiția de promovare $E \geq 5$ & $L \geq 5$

Varianta fizică:

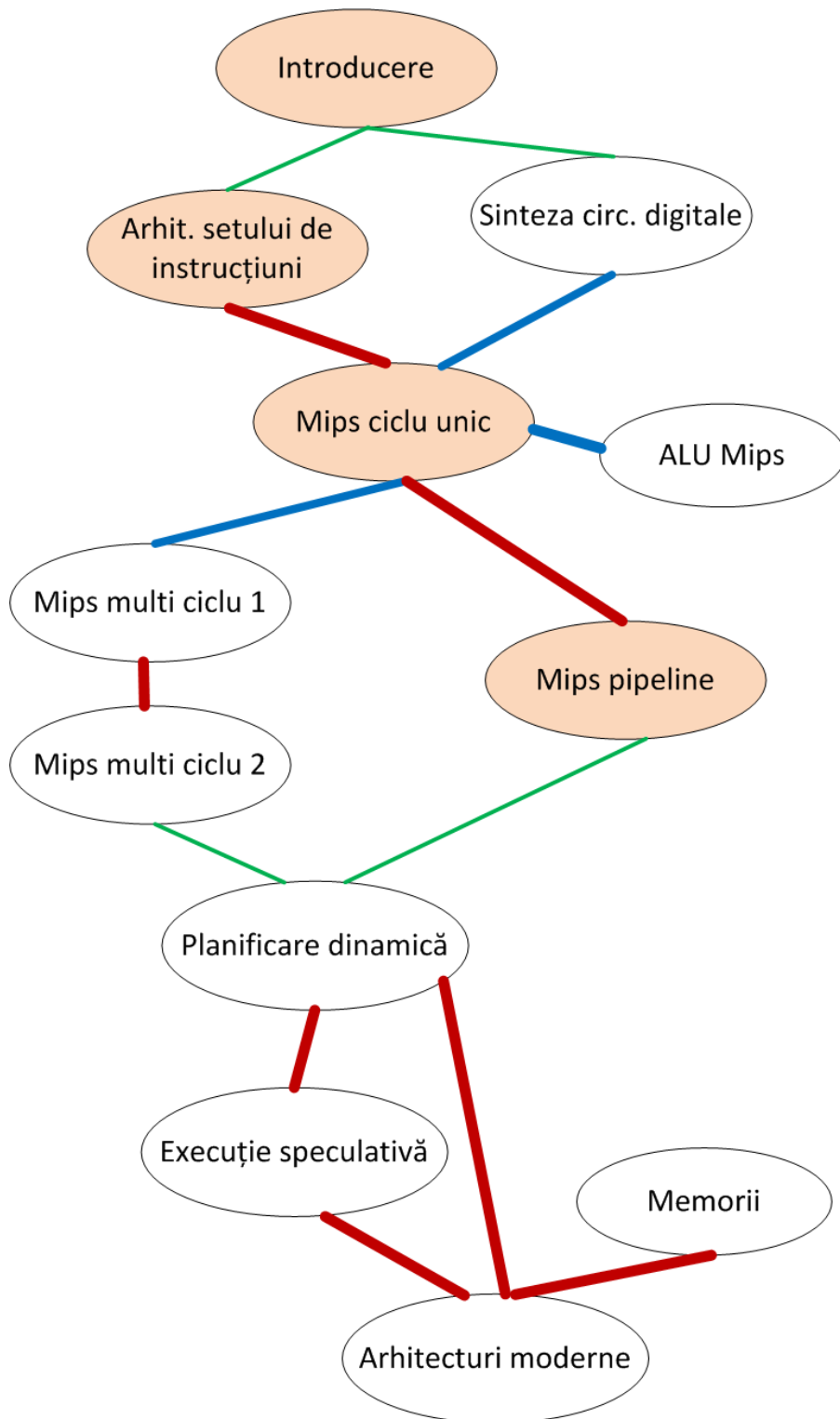
Examenul scris E va avea 2 părți (se face media):

1. Test cu întrebări (grilă sau rezolvări scurte, cu calcule).
2. Probleme bazate pe materialul din curs. O problemă este sigur din procesorul Mips ciclu unic.

Samsung Exynos M5 - microarchitettura



Ordinea / dependența cursurilor



Cunoștințe anterioare: prima parte din materialul obligatoriu [1] sau cunoștințe bune ASDN/PSN

Verde – concepte de bază

Albastru – dependență moderată

Roșu – dependență puternică

Pentru laborator: cursurile cu portocaliu + cursurile când se rezolvă probleme (nu apar explicit)!!!

Pentru examen


- Grilă/mini-probleme: toate cursurile
- probleme: Mips ciclu unic (100%), Alte probleme din orice curs

Bibliografie

1. Florin Oniga, “De la bit la procesor. Introducere în arhitectura calculatoarelor”, Editura U.T. Press, Cluj-Napoca, 2019, ISBN 978-606-737-366-0
2. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 3th edition, ed. Morgan–Kaufmann, 2005
3. D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013, 6th edition 2020
4. D. A. Patterson and J. L. Hennessy, “Computer Organization and Design: A Quantitative Approach”, 5th edition, ed. Morgan-Kaufmann, 2011, 6th edition 2017
5. D. M. Harris, S. L. Harris, *Digital Design and Computer Architecture*, Morgan Kaufmann, San Francisco, 2007
6. D. A. Patterson, J. L. Hennessy, “ORGANIZAREA SI PROIECTAREA CALCULATOARELOR. INTERFATA HARDWARE/SOFTWARE”, Editura ALL, Romania, ISBN: 973-684-444-7
7. Vincent P. Heuring, et al., “Computer Systems Design and Architecture”, *Addison-Wesley*, USA, 1997.
8. MIPS32™ Architecture for Programmers, Volume I: “Introduction to the MIPS32™ Architecture”.
9. MIPS32™ Architecture for Programmers Volume II: “The MIPS32™ Instruction Set”.
10. World Wide Web ...

Nivele de abstracție în sistemele de calcul

Application Software	Programs
Operating Systems	Device Drivers
Architecture	Instructions Registers
Micro-architecture	Datapaths Controllers
Logic	Adders Memories
Digital Circuits	AND gates NOT gates
Analog Circuits	Amplifiers Filters
Devices	Transistors Diodes
Physics	Electrons



Zona de interes pentru cursul de AC:

Digital Circuits, Logic Gates, Register Transfer Level (RTL), Micro-Architecture

Concepte de bază

- Arhitectura – interfața dintre un utilizator și un obiect
- Arhitectura Calculatoarelor
 - Arhitectura Setului de Instrucțiuni (Instruction Set Architecture - ISA)
 - Organizarea Calculatorului (micro-arhitectura)
- ISA - interfața între hardware și programele de nivel jos
- Micro-arhitectura: Componente și conexiunile dintre ele
 - Regs, ALU, Memorie,...
- Aceeași ISA poate avea organizări diferite
 - MIPS: ciclu unic, multi ciclu, pipeline
- O arhitectură particulară se poate implementa prin micro-arhitecturi diferite, cu diferite constrângeri de preț/performanță/putere.
- ISA-uri moderne:
 - IA-32, IA-64, PowerPC, **MIPS**, SPARC, ARM,

Clase de Paralelism și Arhitecturi Paralele

2 tipuri de paralelisme in aplicații:

1. *Data-Level Parallelism (DLP)* – date care pot fi prelucrate în același timp
2. *Task-Level Parallelism (TLP)* – task-uri care pot fi prelucrate independent

Exploatarea paralelismului în 4 moduri:

1. *Instruction-Level Parallelism - ILP* - exploatează paralelismul de date
 - Pipelining, execuție speculativă
2. *Vector Architectures and Graphic Processor Units (GPUs)* - exploatează paralelismul de date prin aplicarea unei singure instrucțiuni la o colecție de date, în paralel
3. *Thread-Level Parallelism* - exploatează paralelismul la nivel de date sau de task, într-un model hardware care permite interacțiuni între thread-uri paralele
4. *Request-Level Parallelism* - exploatează paralelismele între task-uri decuplate, specificate de programator sau de sistemul de operare

Taxonomia lui Flynn

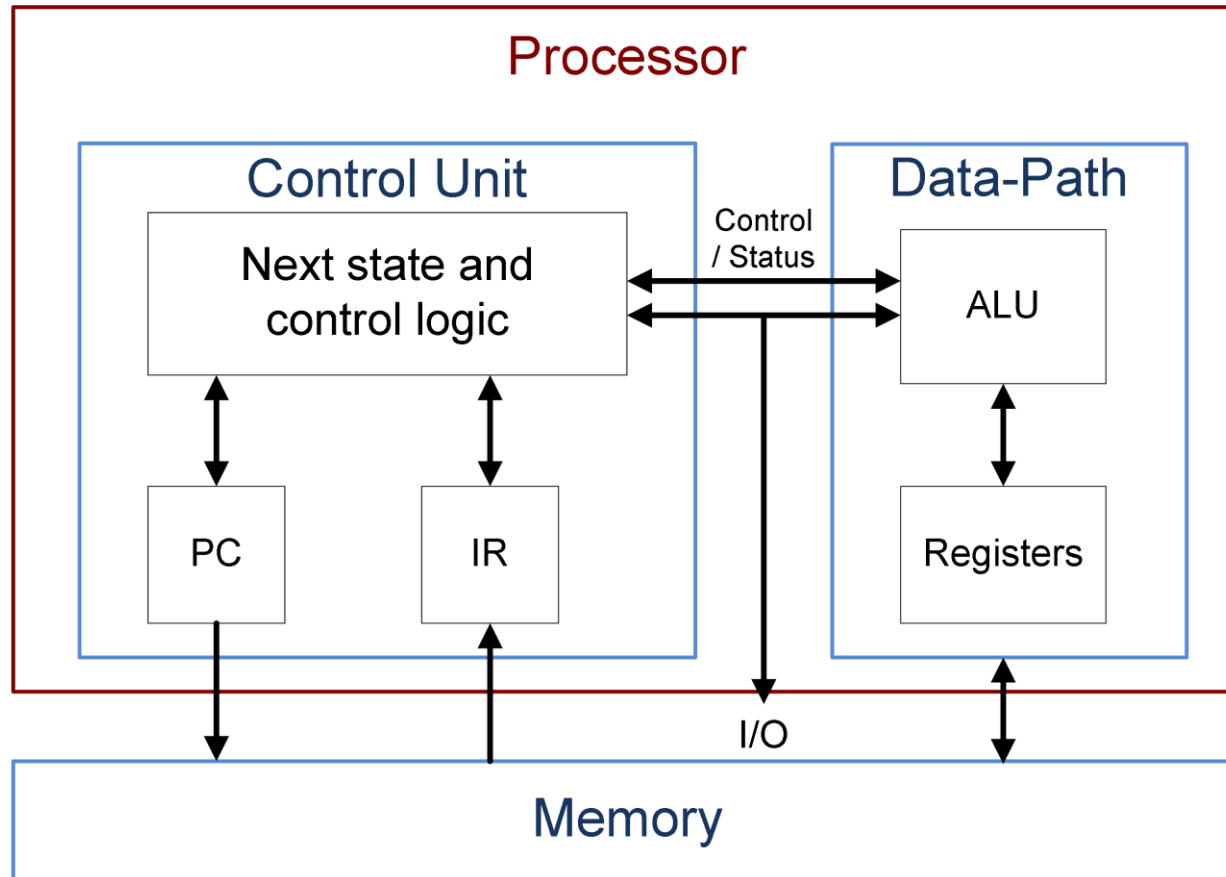
Michael Flynn [1966] – o clasificare simplă pentru arhitecturile de calcul

Flynn's Taxonomy		
	Single Instruction	Multiple Instruction
Single Data	<u>SISD</u>	<u>MISD</u>
Multiple Data	<u>SIMD</u>	<u>MIMD</u>

1. SISD — Uniprocessor, calculator standard, secvențial, poate exploata ILP
2. SIMD — Aceeași instrucțiune se execută de mai multe unități de procesare, asupra diferitelor fluxuri de date, exploatează *data-level parallelism*, arhitecturi de tip Vector (GPU)
3. MISD — foarte rare, redundanță de calcul pentru siguranța calculelor
4. MIMD— Fiecare procesor operează cu instrucțiuni și date proprii, exploatează *Task-Level Parallelism*

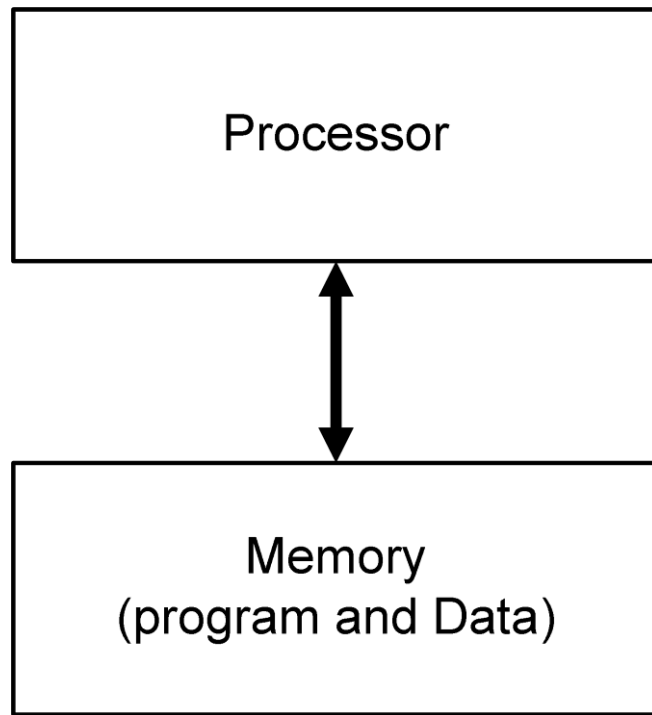
Multe procesoare actuale sunt hibride de SISD, SIMD si MIMD.

Arhitectura generală a unui procesor



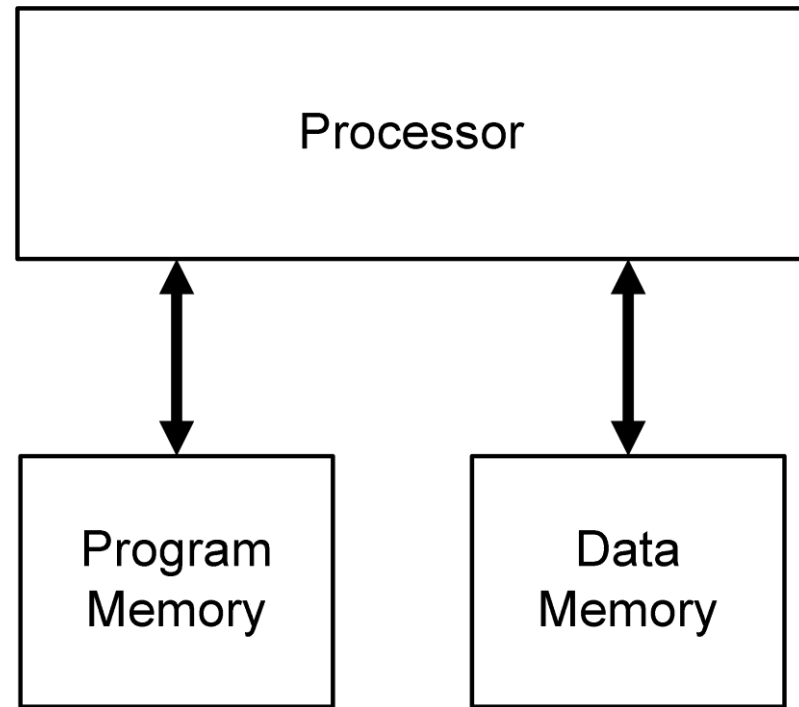
Procesor = Căi de date (Datapath) + Unitate de Control

Arhitecturi clasice uni-procesor



Van Neumann / Princeton

**O memorie unică pentru
instrucțiuni și date**



Harvard

**Memorii separate pentru
instrucțiuni și date**

Evoluția Arhitecturilor

➤ Arhitectura **Complex Instruction Set Computer (CISC)**

- Arhitectura „clasică” înainte de RISC; set de instrucțiuni complex, greu de implementat ca pipeline, număr redus de regiștri, operații ALU cu memorie
 - Acces la memorie prin multe instrucțiuni diferite
 - Multe moduri de adresare
 - Instrucțiuni de lungime variabilă

➤ Arhitectura **Reduced Instruction Set Computer (RISC)**

- Setul de instrucțiuni este „Redus” (mai pragmatic), mai mulți regiștri, mai ușor de implementat ca pipeline, instrucțiuni dedicate pentru lucru cu memoria - arhitectura “load/store”
 - Acces la Memorie - load/store,
 - Procesarea datelor - registru-registru
 - Număr redus de moduri de adresare
 - Instrucțiuni de lungime fixă

Arhitectura Setului de Instrucțiuni (Instruction Set Architecture – ISA)

➤ Interfața între Hardware și programele de nivel jos

Componente ISA:

- Organizarea memoriei, Registre
- Tipuri de date si Structuri de date: Codificări si reprezentări
- Setul de instrucțiuni.
- Formate de instrucțiuni
- Moduri de adresare; pentru instrucțiuni și date
- Tratarea excepțiilor, protecții, I/O
- Etc.

Caracteristici ISA

➤ Clase de ISA

- Majoritatea ISA moderne – arhitecturi GPR - general-purpose register, operanzi ALU – regiștri sau locații de memorie
- Două versiuni de ISA GPR:
 - *register-memory ISA* - x86, x64, operații ALU: reg-reg, sau reg-mem
 - *load-store ISA* - ARM, MIPS, operații ALU: reg-reg, numai Load/Store acces la memorie.

➤ Adresarea Memoriei

- 80x86, ARM, si MIPS, adresare pe byte
- ARM si MIPS, instrucțiunile trebuie să fie aliniate in memorie.
- Acces la un obiect de dimensiune **s octeți** la adresa **Adr** - aliniată dacă **$\text{Adr} \bmod s = 0$**
- 80x86 nu impune aliniere, dar accesul este mai rapid la operanzi aliniați

Caracteristici ISA

➤ Moduri de adresare [2] – cum se identifică / specifică operanzii

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Regs}[R3]$	When a value is in a register.
Immediate	Add R4,#3	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + 3$	For constants.
Displacement	Add R4,100(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[100 + \text{Regs}[R1]]$	Accessing local variables (+ simulates register indirect, direct addressing modes).
Register indirect	Add R4,(R1)	$\text{Regs}[R4] \leftarrow \text{Regs}[R4] + \text{Mem}[\text{Regs}[R1]]$	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1+R2)	$\text{Regs}[R3] \leftarrow \text{Regs}[R3] + \text{Mem}[\text{Regs}[R1] + \text{Regs}[R2]]$	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[1001]$	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[\text{Mem}[\text{Regs}[R3]]]$	If R3 is the address of a pointer p , then mode yields $*p$.
Autoincrement	Add R1,(R2)+	$\begin{aligned} \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \\ \text{Regs}[R2] &\leftarrow \text{Regs}[R2] + d \end{aligned}$	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d .
Autodecrement	Add R1,-(R2)	$\begin{aligned} \text{Regs}[R2] &\leftarrow \text{Regs}[R2] - d \\ \text{Regs}[R1] &\leftarrow \text{Regs}[R1] + \text{Mem}[\text{Regs}[R2]] \end{aligned}$	Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	$\text{Regs}[R1] \leftarrow \text{Regs}[R1] + \text{Mem}[100 + \text{Regs}[R2] + \text{Regs}[R3] * d]$	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Caracteristici ISA

Interpretarea adreselor de memorie

Octeți în registru

3	2	1	0
---	---	---	---

Octeți în memorie

Little Endian

Address	0000	0001	0002	0003
Byte #	0	1	2	3

LSB byte la adresa mem mică

Big Endian

Address	0000	0001	0002	0003
Byte #	3	2	1	0

MSB byte la adresa mem mică

Tipuri si dimensiuni de operanzi

80x86, ARM si MIPS suportă: 8-bit (ASCII character), 16-bit (Unicode character or half word), 32-bit (integer or word), 64-bit (double word or long integer), și IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision).

80x86 suportă si 80-bit floating point (extended double precision).

Caracteristici ISA

Operatori ai setul de instrucțiuni

Tip de operații	Exemple
Arithmetic and logical	Integer arithmetic and logical operations: add, sub, and, or, multiply, divide
Data transfer	Load, stores, move instructions (on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating Point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, multiply, decimal to character conversion
String	String move, compare, search
Graphics	Pixel and vertex operations, compression/decompression operations

Caracteristici ISA

Instrucțiuni de Control – salturi, controlul fluxului de instrucțiuni din program

- Salturi condiționate, salturi necondiționate, apel de proceduri și reveniri.
- Adresare relativă la PC - adresa de salt = offset adunat cu PC
 - Bifurcări MIPS (BEQ, BNE, etc.) testează conținutul registrelor, iar
 - 80x86 și ARM testează biți de condiții (flag), setate ca efecte (colaterale) ale operațiilor aritmetice/logice
- Apelul de procedură ARM și MIPS plasează adresa de revenire într-un registru
- Apelul de procedură 80x86 plasează adresa de revenire în memorie - stiva

Caracteristici ISA

Formatul instrucțiunilor

Doua variante de bază pentru instrucțiuni: lungime *fixă și variabilă*

- ARM si MIPS - instrucțiuni de 32 biți, simplifică decodificarea instrucțiunilor
- 80x86 – lungime variabilă, 1 - 18 octeți.
- Instrucțiunile de lungime variabilă ocupă spațiu mai mic decât cele de lungime fixă
- Numărul registrelor și a modurilor de adresare influențează dimensiunea instrucțiunilor

Caracteristici ISA

Codificarea instrucțiunilor pe biți (formatul)

➤ Variabil (Intel 80x86, VAX)

Operation and no. of operands	Address specifier 1	Address field 1	Address specifier n	Address field n
----------------------------------	------------------------	--------------------	-------	------------------------	--------------------

➤ Fix (Alpha, ARM, MIPS, PowerPC, SPARC)

Operation	Address field 1	Address field 2	Address field 3
-----------	-----------------	-----------------	-----------------

➤ Hibrid (IBM 360/370, MIPS16, Thumb)

Operation	Address specifier	Address field
-----------	-------------------	---------------

Operation	Address specifier 1	Address specifier 2	Address field
-----------	---------------------	---------------------	---------------

Operation	Address specifier	Address field 1	Address field 2
-----------	-------------------	-----------------	-----------------

Clasificarea ISA dptv al localizării operanzilor



[2]

Clasificarea ISA dpdv al localizării operanzilor

Localizarea Operanzilor pentru 4 clase de ISA

(a) **Registrul Top Of Stack (TOS)** – indică operandul de vârf, care se combină cu operandul următor. În timpul operației, operandul (TOS) se îndepărtează de pe stivă. Rezultatul ia locul operandului al doilea, TOS - actualizat, indică rezultatul. Toți operanzii sunt implicați.

(b) **Acumulatorul** este operand de intrare implicit, apoi primește rezultatul operației

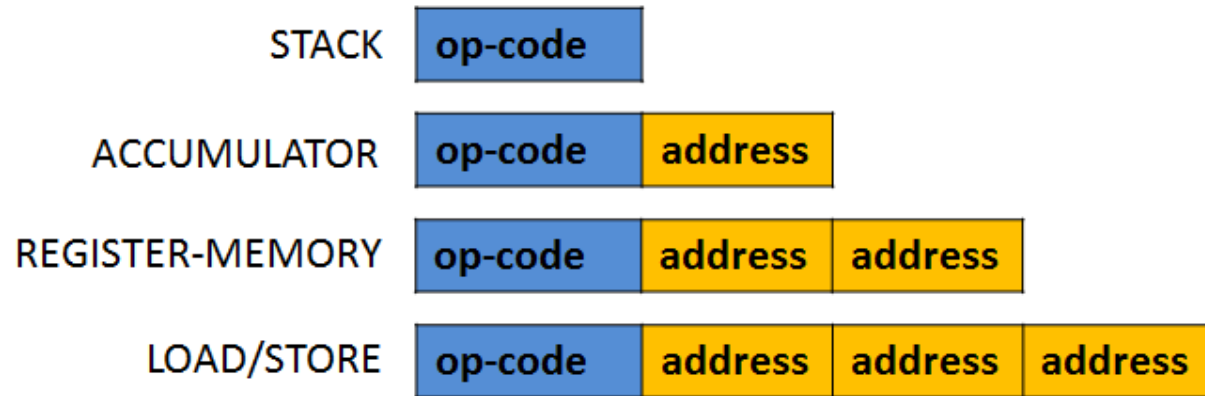
(c) **Registru – Memorie** - operanzii provin din regiștri sau din memorie, rezultatul merge la regiștri.

(d) **Registru – Registru** - operanzi numai din regiștri, ca și pentru stivă

- instrucțiuni separate pentru transfer în/din memorie: push sau pop pentru (a) și load sau store pentru (d).

Clasificarea ISA dpdv al localizării operanzilor

Formatul instrucțiunilor



Secvențe de cod pentru $C = A+B$ - 4 clase de seturi de instrucțiuni

STACK	ACCUMULATOR	REGISTER-MEMORY	LOAD/STORE
Push A	Load A	Load R1, A	Load R1, A
Push B	Add B	Add R1, B	Load R2, B
Add	Store C	Store R1, C	Add R3, R2, R1
Pop C			Store R3, C

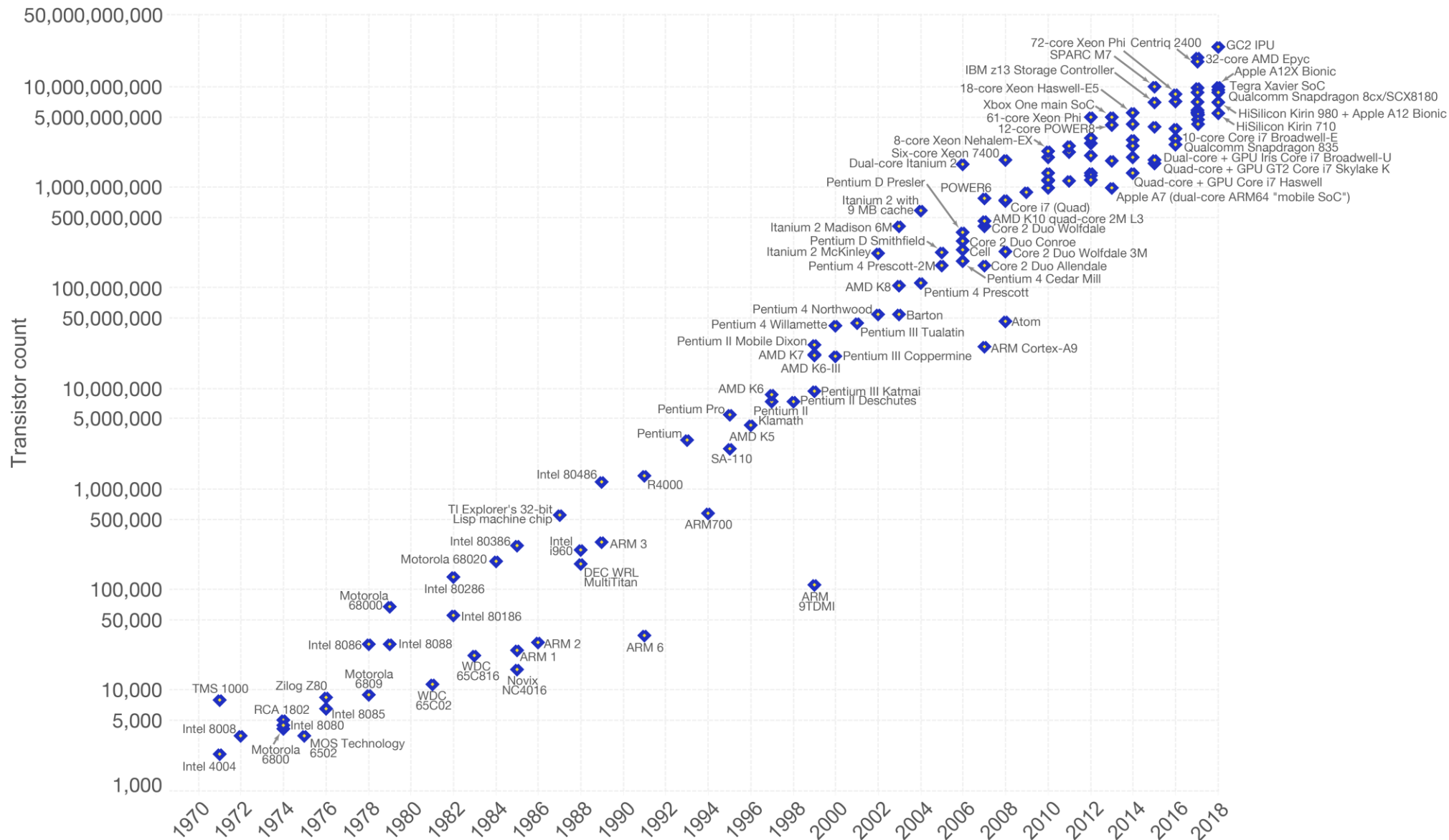
- Instrucțiunea Add are operanzi implicați pentru arhitecturi stivă și acumulator, expliți pentru arhitecturi cu regiștri. A, B și C se găsesc în (locații) memorie

Aspecte legate de tehnologie

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

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.



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)

The data visualization is available at [OurWorldinData.org](https://ourworldindata.org). There you find more visualizations and research on this topic.

Licensed under [CC-BY-SA](#) by the author Max Roser.

Aspecte legate de tehnologie – consum de putere

Putere dinamica

- CMOS chips
 - Puterea consumată per transistor:

$$Power_{dynamic} = \frac{1}{2} \times CapacitiveLoad \times Voltage^2 \times FrequencySwitched$$

- Pentru un task, un ceas mai lent reduce puterea
- Reducerea tensiunii – s-a redus de la 5V la puțin peste 1V în 20 de ani.
- Microprocesoare opresc ceasul pentru module inactive, economie de energie

Puterea Statica

- Important – curenți de scurgere, chiar daca transistorul este inactiv

$$Power_{static} = Current_{static} \times Voltage$$

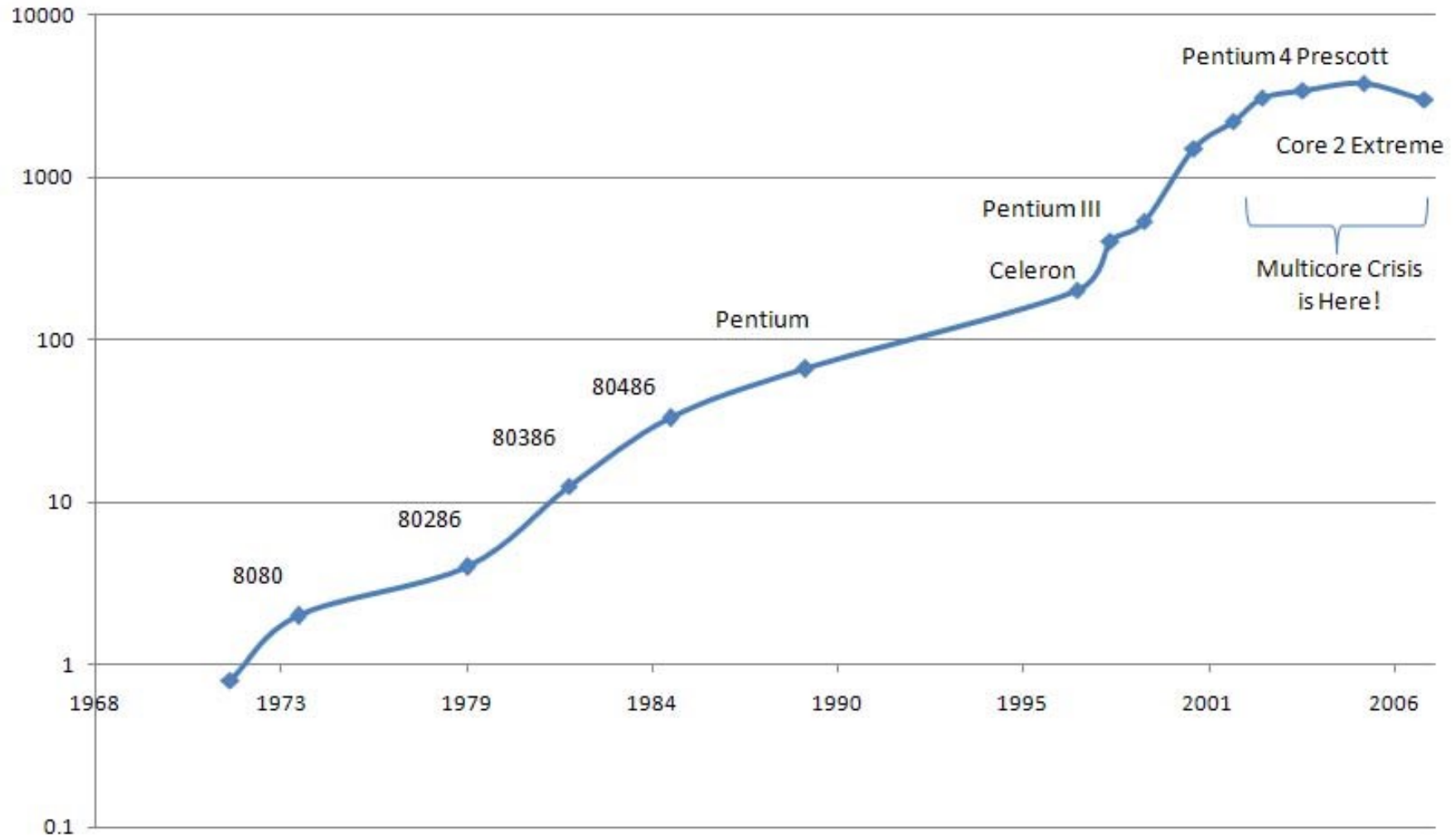
- Proporțională cu numărul dispozitivelor.
- Curentul de scurgere crește cu reducerea mărimii transistoarelor

Aspecte legate de tehnologie – consum de putere

- Limitarea răcirii cu aer a favorizat dezvoltarea de procesoare multiple pe chip, rulând la tensiuni si frecvente reduse
- Primele microprocesoare de 32-biti (Intel 80386) au consumat cca 2 wati, un 3.3 GHz Intel Core i7 consuma 130 wati.
 - Căldura trebuie disipata de la un chip (1,5 cmp) -> s-a atins limita răcirii cu aer

Evoluția frecvenței de ceas

Intel Processor Clock Speed (MHz)



<https://smoothspan.files.wordpress.com/2007/09/clocks speeds.jpg>

2003-2004 – începutul “crizei” care a dus spre procesoarele multi-core

Performanța calculatoarelor – Metrici de evaluare

Bandwidth over Latency

- Bandwidth (lățime de bandă) sau throughput (productivitate) - rezultate produse într-un interval de timp, de ex. megabytes per second pentru transferul pe disk.
 - Important când se rulează mai multe task-uri
- Latency (întârziere) sau response time (timp de răspuns) - durata între pornirea și terminarea unui eveniment, de ex. milliseconds pentru un acces la disk.
 - Important la rularea unui task cu termen limită
- Productivitatea se îmbunătățește mai rapid decât timpul de răspuns

Performanța calculatoarelor – Metrici de evaluare

Timpul CPU pentru un program

IC - the *instruction count* - numărul instrucțiunilor executate

CPI – numărul mediu de *clock cycles per instruction* – cicluri de ceas per instrucțiune

CCT – *clock cycle time* – perioada de ceas.

$$CPUtime = IC \cdot CPI \cdot CCT = \frac{Instructions}{Program} \cdot \frac{Cycles}{Instruction} \cdot \frac{Seconds}{Cycle} = \frac{Seconds}{Program}$$

Performanța Procesorului depinde de 3 caracteristici:

- CCT => Hardware și organizare
- CPI => Organizare și ISA
- IC => ISA și compilator

ISA influențează cele 3 componente ale performanței.

Performanța calculatoarelor – Metrici de evaluare

Ecuția performanței Procesorului

$$Performance_x = \frac{1}{Execution\ time_x}$$

Performanța – inversul Timpului de Execuție

Viteza de execuție a unui program: se măsoară în MIPS – million instructions per second

$$MIPS = \frac{Instruction\ count}{Execution\ time \times 10^6} = \frac{Instruction\ count}{\frac{Instruction\ count \times CPI}{Clock\ rate} \times 10^6} = \frac{Clock\ rate}{CPI \times 10^6}$$

Legea lui Amdahl

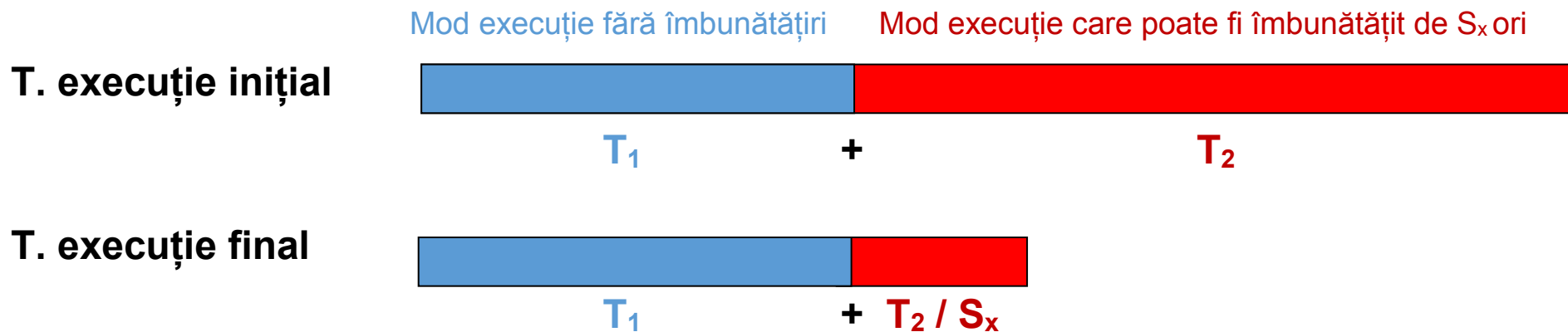
“Îmbunătățirea performanței datorită unui mod mai rapid de execuție este limitată de fracțiunea de timp în care modul respectiv poate fi aplicat.”

Creșterea vitezei - Speedup:

$$Speedup = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

sau,

$$Speedup = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$



Legea lui Amdahl

Speedup depinde de 2 factori:

1. Frațiunea din timpul de calcul în care se beneficiază de îmbunătățire

$$Fraction_{enhanced} = f_x$$

2. Câștigul datorită folosirii îmbunătățirii $Speedup_{enhanced} = S_x$

$$Execution\ time_{new} = Execution\ time_{old} \times \left((1 - f_x) + \frac{f_x}{S_x} \right)$$

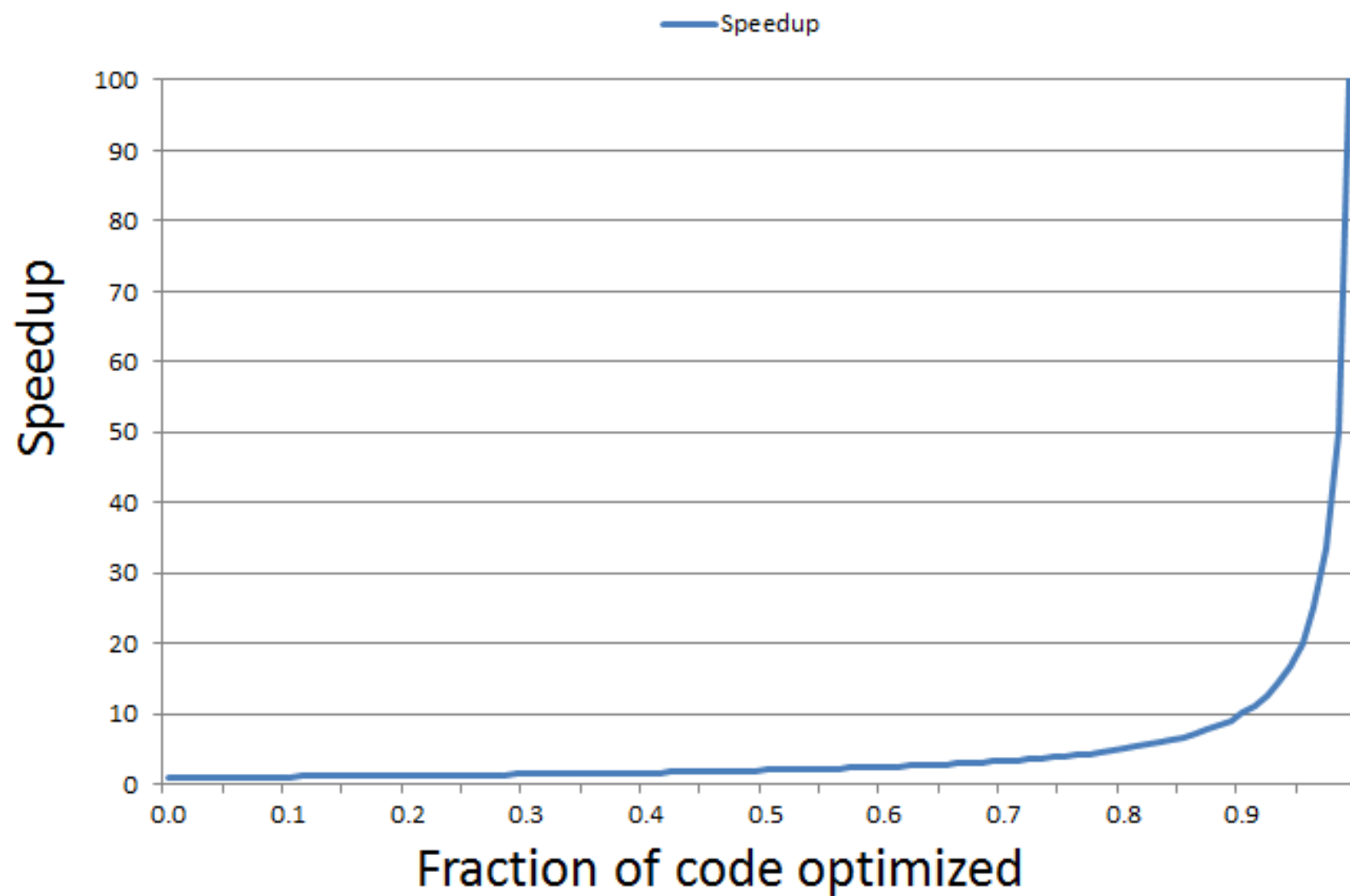
Creșterea vitezei – Speedup – raportul duratelor de execuție:

$$Speedup_{overall} = \frac{Execution\ time_{old}}{Execution\ time_{new}} = \frac{1}{(1 - f_x) + \frac{f_x}{S_x}}$$

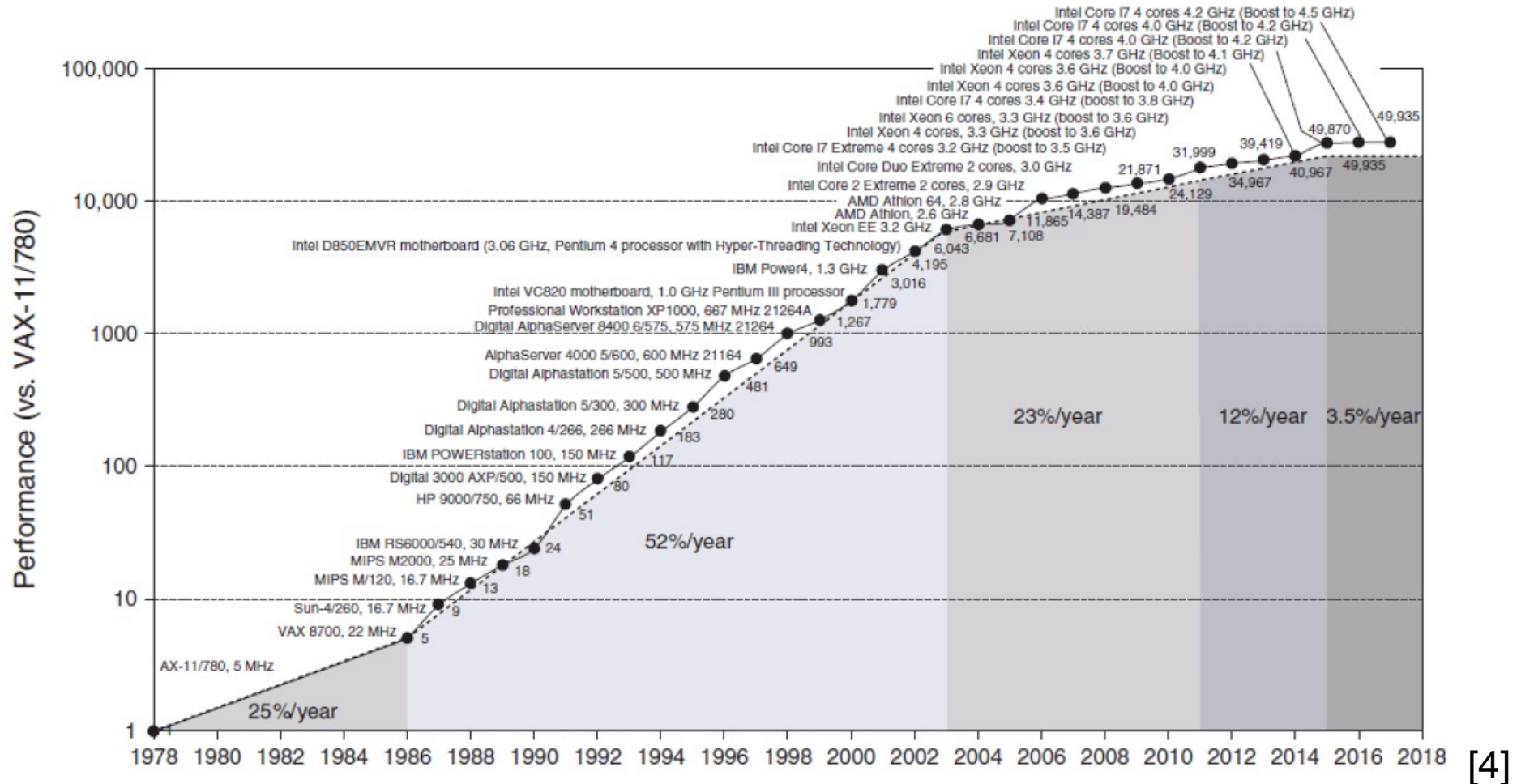
Problemă - Dacă factorul de creștere a vitezei este $S_x = 100$, cum arată Speedup în funcție de frațiunea f_x ?

Legea lui Amdahl

Soluție



Performanța calculatoarelor – Benchmark SPEC



- VAX : 25%/an 1978 – 1986 - datorită tehnologiei
- RISC + x86: 52%/an 1986 - 2002 – idei de arhitectură și organizare
- După 2003, limitare de putere și ILP disponibil, performanța crește lent, în medie cu 22% per an, ajutată mai ales de extinderea spre multi-core, după 2011 +12% per an, după 2015 +3.5% per an

VHDL – De știut (de “ieri” !)

➤ Tipuri de circuite

- **Circuite combinaționale** – schimbarea valorilor de intrare duce la schimbarea ieșirii
- **Circuite secvențiale** – schimbarea este posibilă doar la momente discrete de timp (ex. pe front crescător de ceas) – au “memorie”...

➤ Circuite de bază

- **Porți logice**
- **Multiplexoare**
- **Decodificatoare**
- **Bistabile D (D-Latches and D-Flip-Flops)**
- **Numărătoare**
- **Memorii**

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

Reguli de scriere în VHDL

- Nu veți crea o nouă entitate pentru orice circuit. Nu creați entități separate pentru circuite simple (de bază): porți logice, bistabile, multiplexoare, numărătoare, decodificatoare, etc.
- Nu abuzați de descrierea structurală până la nivel de granularitate de porți logice!!
- Bazați-vă în principal pe descrierea comportamentului.
- Veți crea entități noi doar pentru părțile semnificative ale proiectului (acest lucru se menționează explicit în tutorialele de la laborator).

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Declaraarea unui semnal de 1 bit

```
signal sig_name : std_logic := '0';
```

➤ Declaraarea unui semnal de N biți

```
signal sig_name: std_logic_vector(N-1 downto 0): ="00....0";
```

➤ Inițializare, exemple

```
semnal 16 biți      "0000000000000000";
```

```
semnal 32 biți      x"00000000";
```

```
semnal N biți      (others => '0');
```

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Porți logice



NOT

$O \leq \text{not } A;$



AND

$O \leq A \text{ and } B;$



OR

$O \leq A \text{ or } B;$



NAND

$O \leq A \text{ nand } B;$



NOR

$O \leq A \text{ nor } B;$



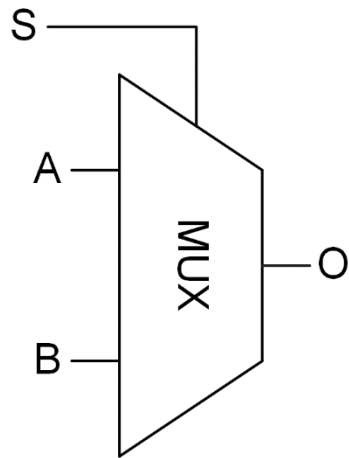
XOR

$O \leq A \text{ xor } B;$

➤ FĂRĂ entitate suplimentară, doar semnalele necesare se declară în entitatea principală!

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Multiplexor 2:1

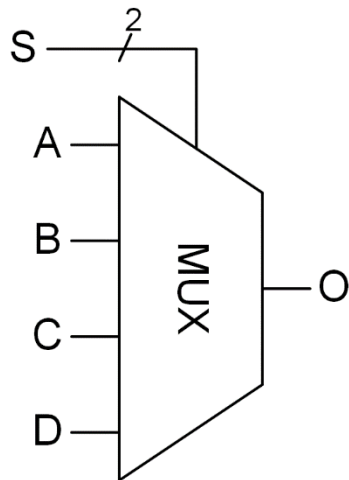


FĂRĂ entitate suplimentară,
doar semnalele necesare se
declară în entitatea principală!

```
O <= A when S = '0' else B;
```

```
process(S, A, B)
begin
    if(S = '0') then
        O <= A;
    else
        O <= B;
    end if;
end process;
```

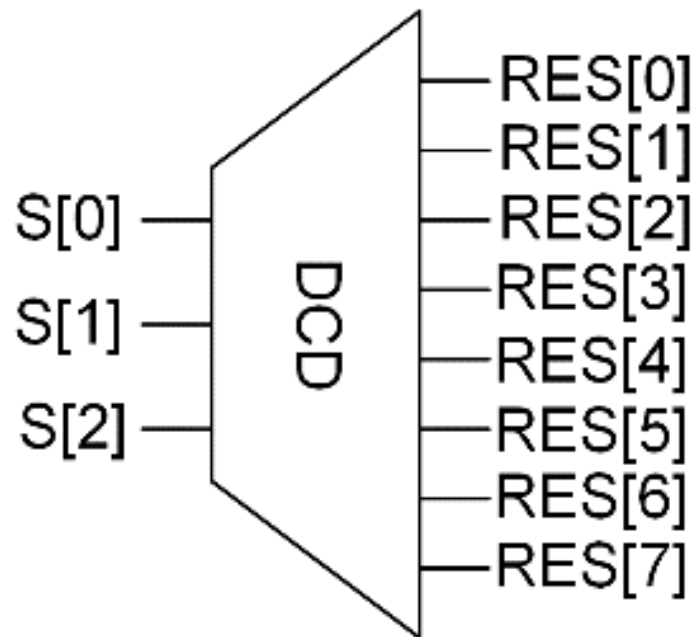
➤ Multiplexor 4:1



```
process(S, A, B, C, D)
begin
    case S is
        when "00" => O <= A;
        when "01" => O <= B;
        when "10" => O <= C;
        when others => O <= D;
    end case;
end process;
```

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Decodificator 3:8



```
process(S)
begin
    case S is
        when "000" => RES <= "00000001";
        when "001" => RES <= "00000010";
        when "010" => RES <= "00000100";
        when "011" => RES <= "00001000";
        when "100" => RES <= "00010000";
        when "101" => RES <= "00100000";
        when "110" => RES <= "01000000";
        when others => RES <= "10000000";
    end case;
end process;
```

FĂRĂ entitate suplimentară,
doar semnalele necesare se
declară în entitatea principală!

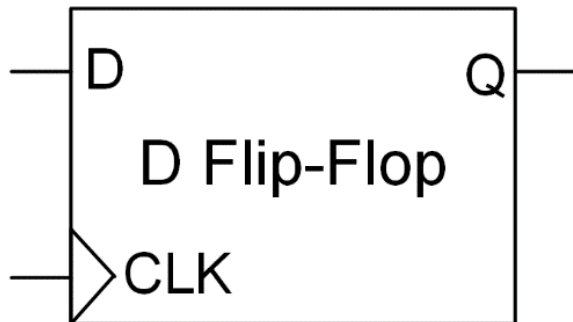
VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Bistabil D-latch (zăvor)



```
process(G, D)
begin
    if(G = '1') then
        Q <= D;
    end if;
end process;
```

➤ Bistabil D sincron (D Flip-Flop) – referit frecvent ca registru!

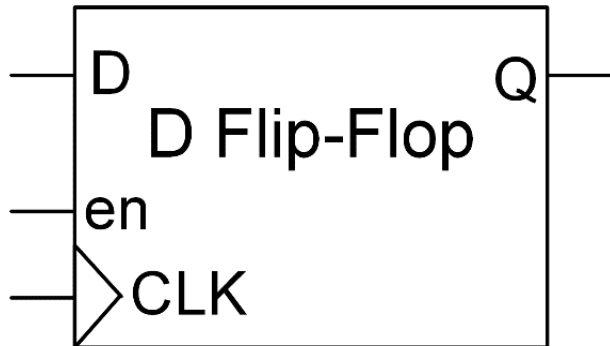


```
process(clk)
begin
    if rising_edge(clk) then
        Q <= D;
    end if;
end process;
```

FĂRĂ entitate suplimentară,
doar semnalele necesare se
declară în entitatea principală!

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Bistabil D sincron cu activare/validare (D Flip-Flop with enable)



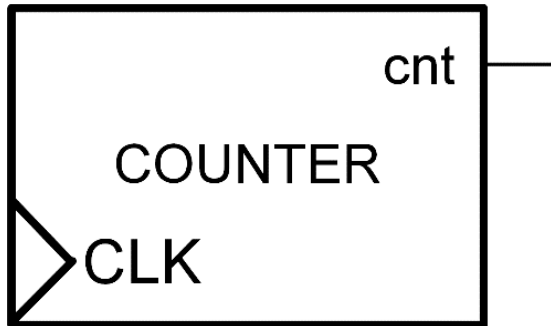
```
process(clk)
begin
    if rising_edge(clk) then
        if en = '1' then
            Q <= D;
        end if;
    end if;
end process;
```

- rising_edge(clk) este echivalent cu clk'event and clk = '1'
- Nu folosiți rising_edge(clk) and en = '1'!

FĂRĂ entitate suplimentară, doar semnalele necesare se declară în entitatea principală!

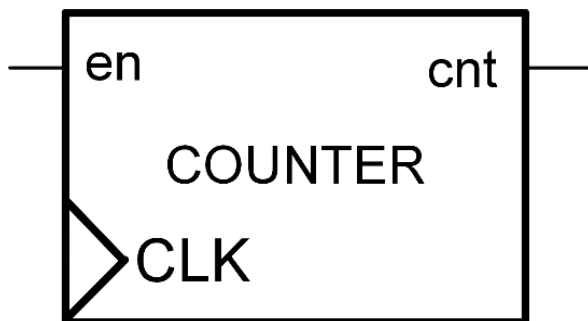
VHDL – De știut (de mâine!!!!!!!!!!!!!!)

➤ Numărător crescător



```
process(clk)
begin
    if rising_edge(clk) then
        cnt <= cnt + 1;
    end if;
end process;
```

➤ Numărător crescător cu semnal de activare / validare

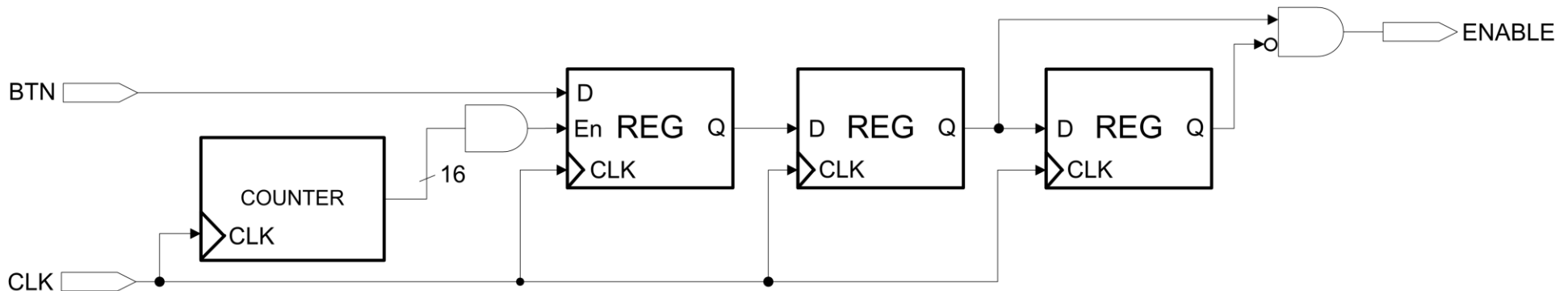


```
process(clk)
begin
    if rising_edge(clk) then
        if en = '1' then
            cnt <= cnt + 1;
        end if;
    end if;
end process;
```

FĂRĂ entitate suplimentară, doar semnalele necesare se declară în entitatea principală!

VHDL – De știut (de mâine!)

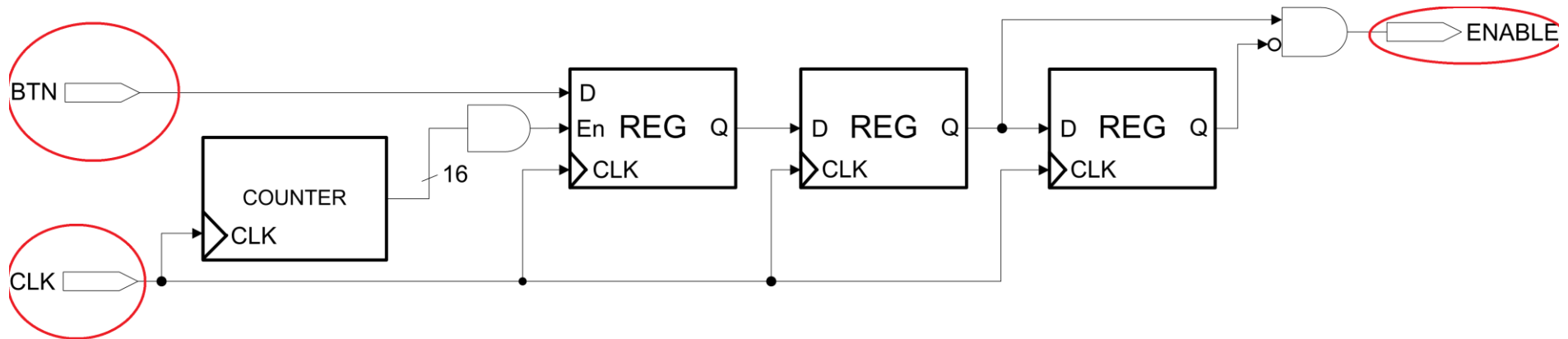
- Cum „traducem” o schemă de circuit în descriere VHDL ? Studiu de caz: generatorul de monoimpuls.



Urmează descrierea metodologiei pe pași! Toată descrierea se face în același fișier sursă.

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

1. Identificăm porturile de intrare/ieșire pentru a descrie entitatea (dacă e cazul).



...declarare librării...

```
entity MPG is
  Port ( en : out STD_LOGIC;
        input : in STD_LOGIC;
        clock : in STD_LOGIC);
end MPG;
```

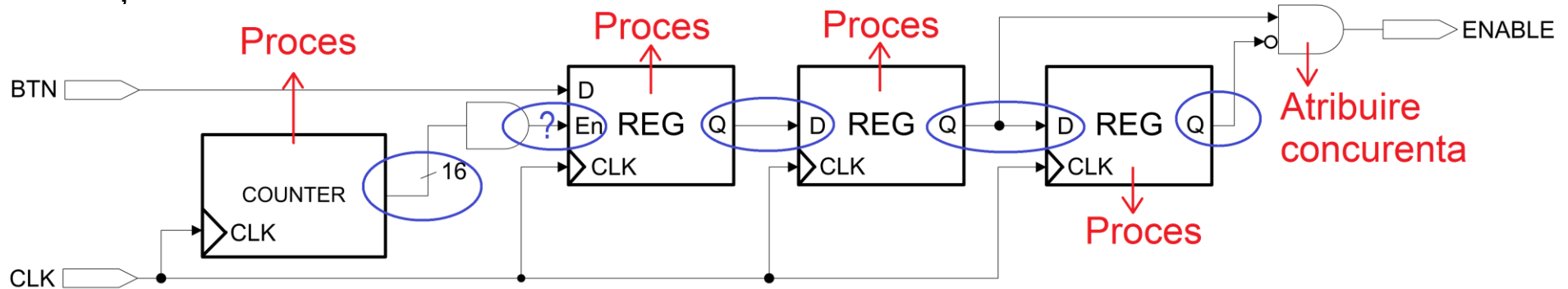
architecture Behavioral of MPG is

...declarare semnale...

```
Begin
...descriere comportament...
end Behavioral;
```

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

2. Identificăm/declarăm semnalele interne care trebuie declarate în arhitectura entității.



Atenție la redundanță: ex. declarăm semnal pentru D la cei 3 regiștri? **NU!**

entity MPG is

```

.....
end MPG;

```

architecture Behavioral of MPG is

```

signal count_int : std_logic_vector(31 downto 0) :=x"00000000";
signal Q1 : std_logic;
signal Q2 : std_logic;
signal Q3 : std_logic;

```

Begin

...descriere comportament (**procesе si atribuirі concurente**)...

end Behavioral;

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

3. Descriem pe rând circuitele individuale cu procese sau atribuiri concurente.

Unde? În arhitectură, după begin. În ce ordine? NU CONTEAZĂ!

Begin

en <= Q2 AND (not Q3);

process (clock)

begin

if clock'event and clock='1' then

count_int <= count_int + 1;

end if;

end process;

process (clock)

begin

if clock'event and clock='1' then

if count_int(15 downto 0) = "1111111111111111" then

Q1 <= input;

end if;

end if;

end process;

process (clock)

begin

if clock'event and clock='1' then

Q2 <= Q1;

Q3 <= Q2;

end if;

end process;

end Behavioral;

VHDL – De știut (de mâine!!!!!!!!!!!!!!)

Cum se sintetizează descrierea în VHDL? Dar legăturile între componente?

```
Begin
en <= Q2 AND (not Q3);
process (clock)
begin
if clock='1' and clock'event then
count_int <= count_int + 1;
end if;
end process;
process (clock)
begin
if clock'event and clock='1' then
if count_int(15 downto 0) = "1111111111111111" then
Q1 <= input;
end if;
end if;
end process;
process (clock)
begin
if clock'event and clock='1' then
Q2 <= Q1;
Q3 <= Q2;
end if;
end process;
end Behavioral;
```



Poarta SI



Numărătorul



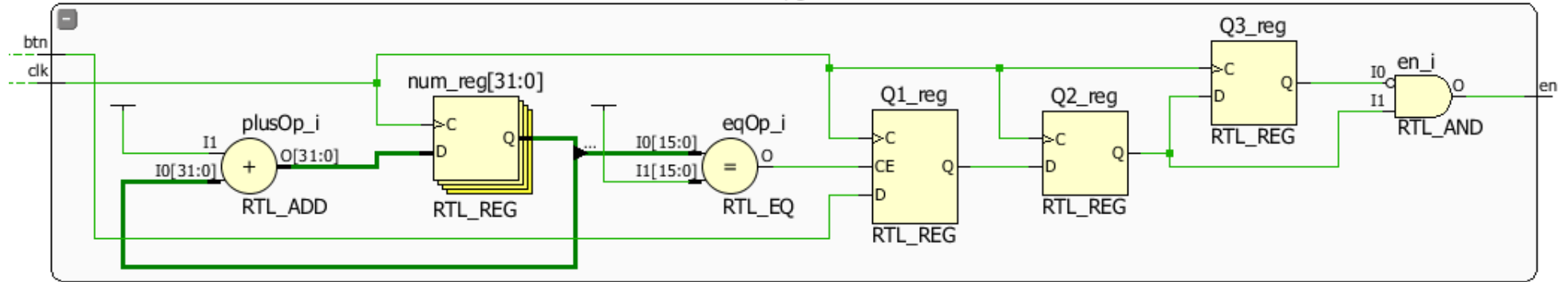
Primul Reg cu
validare (poarta SI
pe 16b inclusă)



Reg 2 si 3

mpg1

mpg



<https://149664534.v2.pressablecdn.com/wp-content/uploads/2018/12/forgetting-v1.png>

<https://149664534.v2.pressablecdn.com/wp-content/uploads/2022/01/learning-v1-updated.png>