

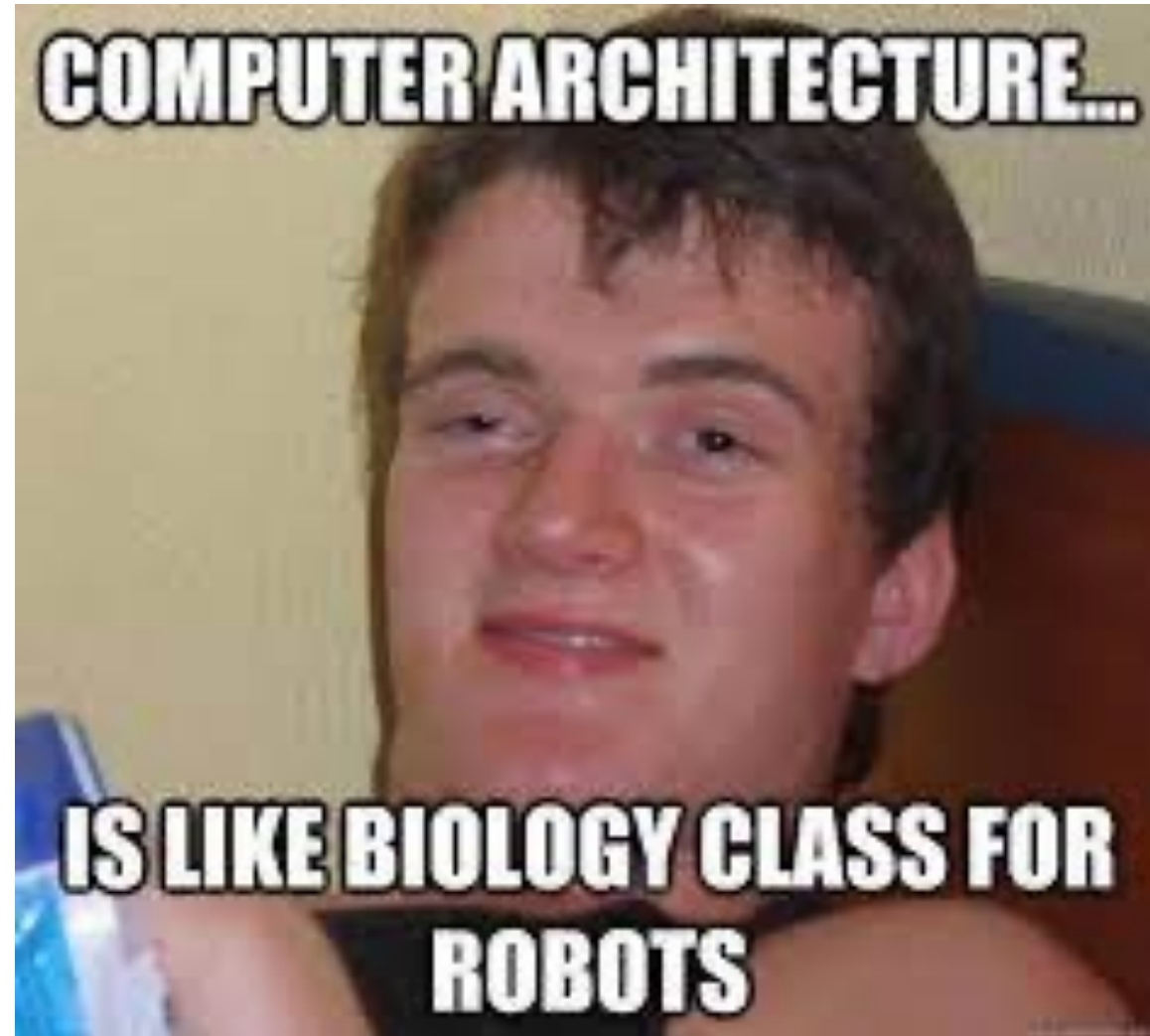
# Sisteme Încorporate

## Cursul 2

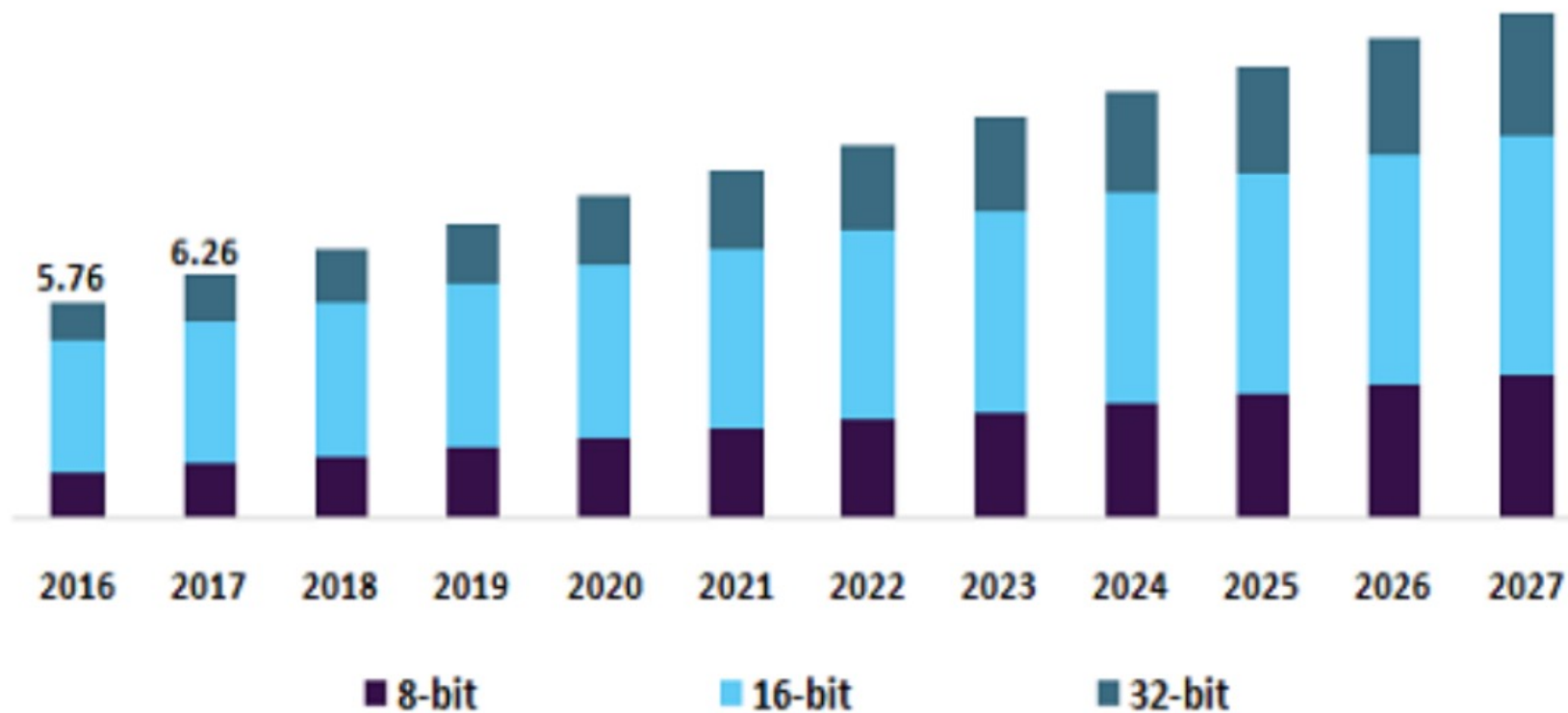
### Arhitectura Embedded

Facultatea de Automatică și Calculatoare  
Universitatea Politehnica București

# Comic of the day

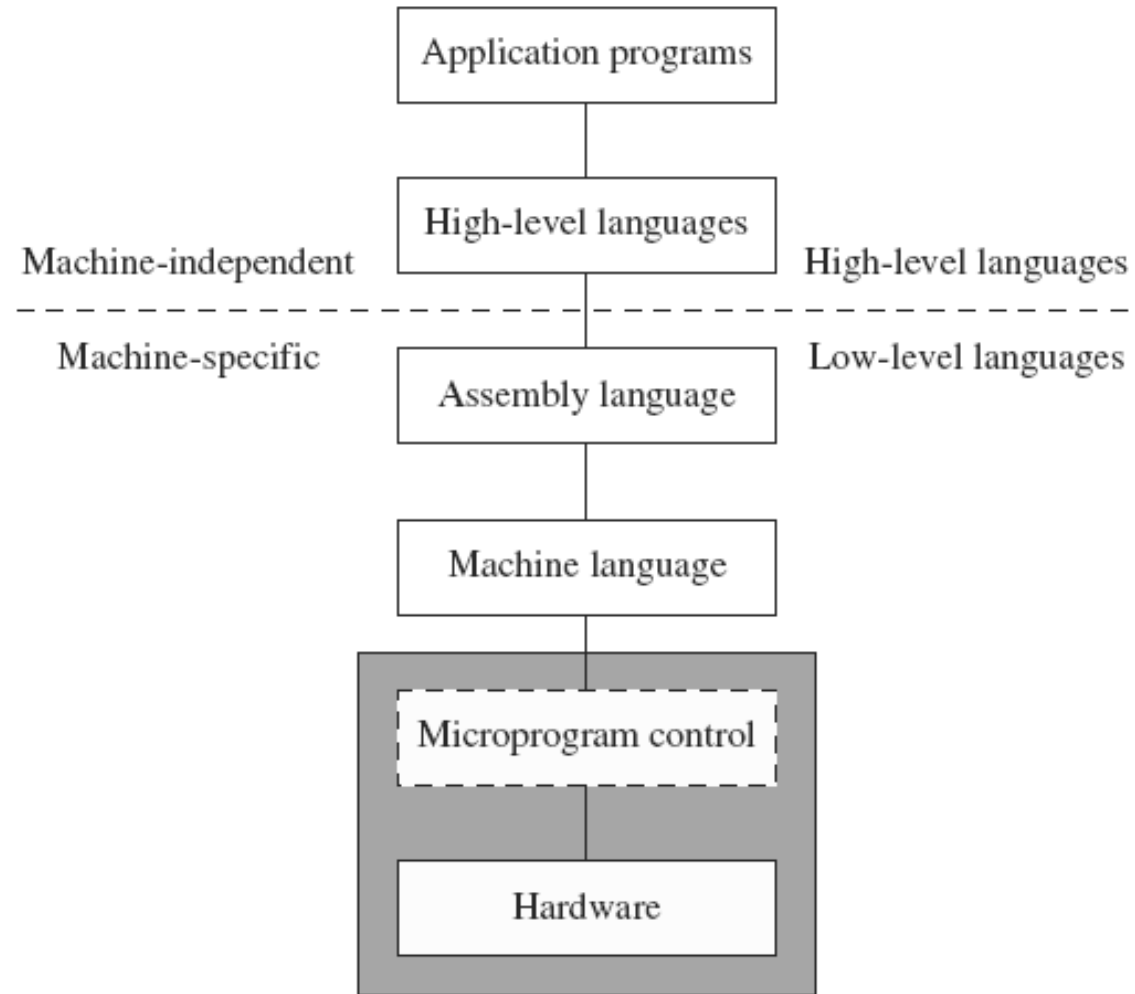


## North America microcontroller market size, by product, 2016 - 2027 (USD Billion)

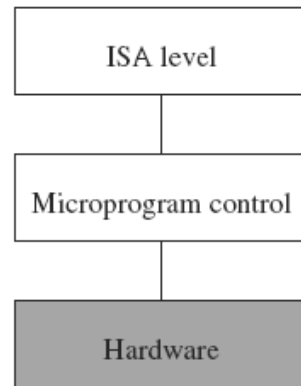


- Folosim arhitectura setului de instrucțiuni (ISA) pentru a abstractiza funcționarea internă a unui procesor.
- ISA definește “personalitatea” unui procesor
  - Cum funcționează procesorul
  - Ce fel de instrucțiuni execută
  - Care este interpretarea dată instrucțiunilor
- Folosind ISA bine definit, putem descrie un procesor la nivel logic
- Chipuri cu structură hardware total diferită pot folosi aceeași ISA
  - Ex: Intel Pentium, Celeron și Xeon folosesc IA-32

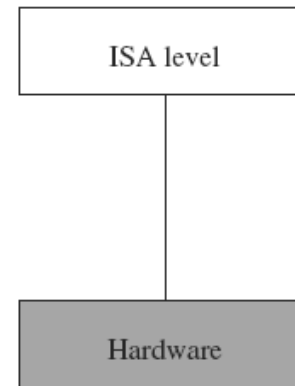
# Sistem de calcul – programmer's view



- Există două filozofii opuse de design pentru un procesor
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)
- Dar, ca de obicei, lucrurile nu sunt atât de simple
- Cum se pot deosebi din perspectiva ISA?



(a) CISC implementation



(b) RISC implementation

# Numărul de adrese pe instrucțiune

- O caracteristică ISA ce definește arhitectura procesorului o constituie numărul de adrese folosite într-o instrucțiune
- Majoritatea operațiilor sunt de două tipuri:
  - Unare (not, and, xor etc.)
  - Binare (add, sub, mul ...)
- Majoritatea procesoarelor folosesc trei adrese: operanzi și destinație
  - $\text{add}(\text{dest}, \text{op1}, \text{op2}) \rightarrow \text{dest} = \text{op1} + \text{op2}$
  - $\text{mul}(\text{dest}, \text{op1}, \text{op2}) \rightarrow \text{dest} = \text{op1} * \text{op2}$
- Putem reduce numărul adreselor la două
  - $\text{add}(\text{op1}, \text{op2}) \rightarrow \text{op1} = \text{op1} + \text{op2}$
- Sau chiar la una...sau zero adrese.

# Three address machines

- Instrucțiunea conține toate cele trei adrese
- Exemplu:

$$A = B + C * D - E + F + A$$



mult T,C,D ;	T = C*D
add T,T,B ;	T = B + C*D
sub T,T,E ;	T = B + C*D - E
add T,T,F ;	T = B + C*D - E + F
add A,T,A ;	A = B + C*D - E + F + A

4  
4  
4  
4  
4

Memory access



# Two address machines

- În exemplul anterior, aproape toate instrucțiunile foloseau de două ori aceeași adresă.
  - De ce să nu folosim acea adresă pe post de sursă și destinație?
- IA-32 folosește aceasta schemă

$$A = B + C * D - E + F + A$$



load T,C ;	T = C
mult T,D ;	T = C*D
add T,B ;	T = B + C*D
sub T,E ;	T = B + C*D - E
add T,F ;	T = B + C*D - E + F
add A,T ;	A = B + C*D - E + F + A

3  
4  
4  
4  
4  
4

Memory access

# One address machines

- Folosesc un registru special pentru a încărca operandul și a stoca rezultatul (registru acumulator)
- Au fost printre primele arhitecturi de procesoare
  - Necesită cel mai mic spațiu de memorie dintre toate arhitecturile

$$A = B + C * D - E + F + A$$



load C ;	load C into the accumulator
mult D ;	accumulator = C*D
add B ;	accumulator = C*D+B
sub E ;	accumulator = C*D+B-E
add F ;	accumulator = C*D+B-E+F
add A ;	accumulator = C*D+B-E+F+A
store A ;	store the accumulator contents in A

2  
2  
2  
2  
2  
2  
2

Memory access

# Zero address machines

- Folosesc stiva pentru stocarea operanzilor și a rezultatului – stack machines

$$A = B + C * D - E + F + A$$



push E ;	<E>	2
push C ;	<C, E>	2
push D ;	<D, C, E>	2
mult ;	<C*D, E>	1
push B ;	<B, C*D, E>	2
add ;	<B+C*D, E>	1
sub ;	<B+C*D-E>	1
push F ;	<F, B+D*C-E>	2
add ;	<F+B+D*C-E>	1
push A ;	<A, F+B+D*C-E>	2
add ;	<A+F+B+D*C-E>	1
pop A ;	<>	2

Memory access

# Comparație

Formatul instrucțiunii				Len	Accese la memorie
8 bits	5 bits	5 bits	5 bits	23	20
Opcode	Rdest	Rsrc1	Rsrc2		
3-address format					
8 bits	5 bits	5 bits		18	23
Opcode	Rdest/Rsrc1	Rsrc2			
2-address format					
8 bits	5 bits			13	14
Opcode	Rdest/Rsrc2				
1-address format					
8 bits				8	19
Opcode					
0-address format					

- Arhitectura dominantă pe piața PC-urilor este (încă) cea Intel.
- Procesoarele folosesc un set complex de instrucțiuni (CISC).
- Care este motivația actuală?
  - Închiderea prăpastiei semantice (semantic gap).
- Efectele secundare ale acestei decizii sunt greu de ignorat.

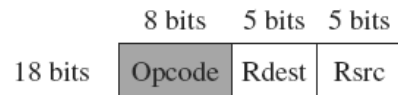
- De ce RISC e mai bun decat CISC în embedded (si nu numai) ?
  - La începutul erei calculatoarelor, procesoarele aveau un design simplu și un număr limitat de funcții
  - În timp, funcțiile au devenit tot mai numeroase
  - Adăugarea de instrucțiuni complexe a dus la complicarea design-ului procesorului
  - Mai multe tranzistoare pe chip:
    - costuri ridicate de producție
    - consum foarte mare de energie
    - probleme cu disiparea căldurii

Regula 80-20 : 80% din timp se folosesc 20% din instrucțiunile unui procesor.

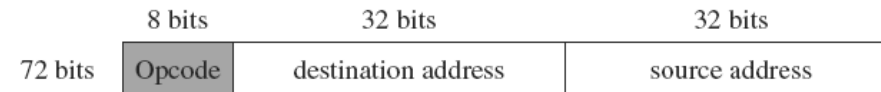
1. Multe instrucțiuni CISC sunt nefolosite de către programatori
2. Majoritatea instrucțiunilor complexe pot fi “sparte” în mai multe instrucțiuni simple

- Exemplu. Înmulțirea a două numere:

RISC	CISC
LOAD R1, D1 LOAD R2, D2 PROD R1, R2 STORE D1, R1	MULT D1, D2



1. Arhitectura Load & Store
2. Fiecare instrucțiune se execută într-un singur ciclu de ceas
3. După execuție registrele R1 și R2 rămân inițializate
4. Mai multe linii de cod -> memorie de program mai mare



1. Load/Store se execută transparent
2. Instrucțiune complexă – poate să dureze mai multe cicluri de ceas
3. După execuție registrele generale sunt aduse la zero
4. O singură linie de cod



# CPU Performance Equation

$$\begin{aligned} \text{CPU time} &= \text{CPU Clock Cycles} \times \text{Clock cycle time} \\ \text{CPU time} &= \text{Instruction Count} \times \text{Cycles Per Instruction} \times \text{Clock cycle time} \\ \text{CPU time} &= \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}} \end{aligned}$$

ISA,  
Compiler  
Tehnologie

Organizare,  
ISA

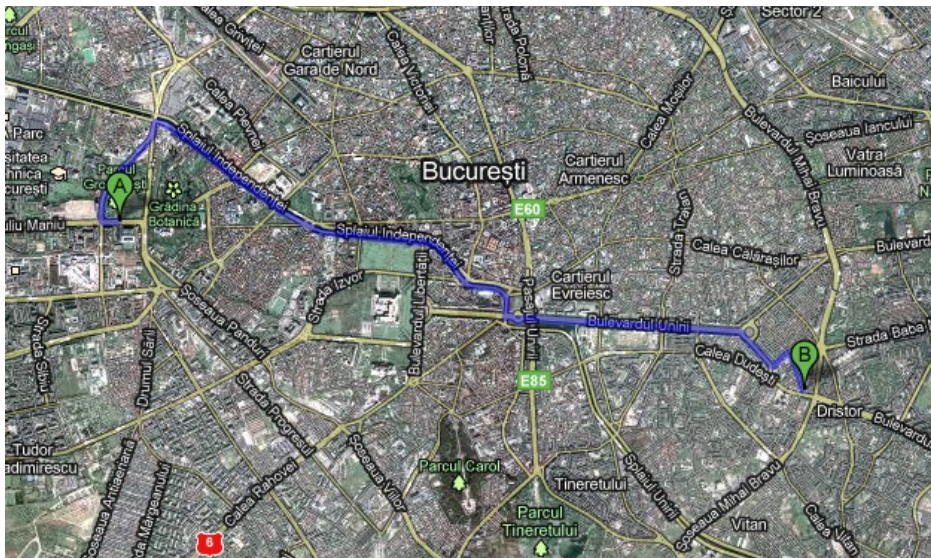
Hardware  
Tehnologie,  
Organizare

A.K.A. The “iron law” of performance

# Analogie cu o mașină

- Trebuie sa ajung de la Poli la Mall Vitan
  - “Clock Speed” = 3500 RPM
  - “CPI” = 5250 rotații/km sau 0.19 m/rot
  - “Insts” = 8.4km

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$



$$8400\text{m} \times \frac{1 \text{ rotație}}{0.19 \text{ m}} \times \frac{1 \text{ minut}}{3500 \text{ rotații}}$$

= 12.6 minute

# Versiunea CPU

- Un program are în total 33 miliarde de instrucțiuni
- CPU procesează instrucțiunile cu 2 cicli pe instr.
- Clock speed -> 3GHz

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

Câteodata se dă perioada  
ceasului în loc de  
Frecvență(ex. cycle = 333 ps)

Câteodată IPC în loc de CPI

= 22 secunde

# CPU Performance Equation (2)

CPU time = CPU Clock Cycles  $\times$  Clock cycle time

$$\text{CPU time} = \left( \sum_{i=1}^n \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

Pentru fiecare tip  
de instrucțiune

De câți cicli este nevoie  
pentru a executa  
instrucțiunea de acest tip

Câte instrucțiuni de  
acest tip sunt în  
program

# CPU performance cu diferite tipuri de instrucțiuni

Tip Instructiune	Frecventa	CPI
Integer	40%	1.0
Branch	20%	4.0
Load	20%	2.0
Store	10%	3.0

$$\text{CPU time} = \left( \sum_{i=1}^n IC_i \times CPI_i \right) \times \text{Clock cycle time}$$

Total Insts = 50B, Clock speed = 2 GHz

# Cum comparăm două procesoare?

- “X este de n ori mai rapid ca Y”

$$\frac{\text{Timp executie}_Y}{\text{Timp executie}_X} = n$$

- “Productivitatea lui X este de n ori cea a lui Y”

$$\frac{\text{Taskuri pe unit. de timp}_X}{\text{Taskuri pe unit. de timp}_Y} = n$$

# Lucrurile nu sunt însă așa de ușoare

- “X este de n ori mai rapid ca Y pentru A”

$$\frac{\text{Timpe de executie al app. A pe masina Y}}{\text{Timpe de executie al app. A pe masina X}} = n$$

- Dar ce se întâmplă pentru aplicații diferite?  
(sau chiar pentru părți ale aceleiași aplicații)
  - X este de 10 ori mai rapid decât Y rulând A, și de 1.5 ori rulând B, dar Y este de 2 ori mai rapid decât X rulând C, și de trei ori rulând D, și...

Deci, X e mai performant decât Y?

Pe care l-ai cumpăra?

- Folosim o medie aritmetică
  - Timpul mediu de execuție
  - Acordă o pondere mai mare programelor care rulează mai mult
- Medie aritmetică ponderată
  - Programele mai importante pot fi evidențiate mai bine
  - Dar ce folosim pentru ponderi?
  - Ponderi diferite vor face ca mașini diferite să arate mai bine în clasament





# RISC vs. CISC: Concluzie

RISC	CISC
<ol style="list-style-type: none"><li>1. Pune accent pe software</li><li>2. Instrucțiuni reduse într-un singur ciclu de ceas</li><li>3. Load &amp; Store ca instrucțiuni separate</li><li>4. Cod cu multe instrucțiuni</li><li>5. Complexitate redusă - număr mic de tranzistoare pe chip (lasă loc de periferice)</li><li>6. Necesită un spațiu mărit de memorie pentru program și date</li></ol>	<ol style="list-style-type: none"><li>1. Accent pe hardware</li><li>2. Instrucțiuni complexe în unul sau mai mulți cicli</li><li>3. Load/Store încorporate în instrucțiunea complexă</li><li>4. Cod de lungime redusă</li><li>5. Complexitate mărită – număr mare de tranzistoare alocate executării instrucțiunilor complexe</li><li>6. Nu are nevoie de foarte multă memorie</li></ol>

- În trecut piața era dominată de procesoare CISC
- Ce a blocat dezvoltarea RISC?
  - Tehnologiile existente în trecut
  - Lipsa software (compilatoare doar pentru x86)
  - Prețul ridicat al memoriilor
  - Lipsa de interes a pieței
  - Intel

- Ce se întâmplă în prezent
  - Granița dintre cele două arhitecturi este neclară
  - Majoritatea procesoarelor au o arhitectură hibridă
    - Intel Pentium: interpretor CISC peste un nucleu RISC
  - **Procesoarele CISC nu sunt potrivite pentru embedded**
- RISC a monopolizat domeniul Embedded
  - Cost redus al memoriei
  - Compilatoare eficiente
  - Costuri mici de fabricație
  - Consum redus de energie
  - Eficiență crescută

- Arhitecturi RISC
  - ARM
  - RISC-V
  - Cadence (Xtensa)
  - MIPS
  - Power Architecture (IBM, Freescale)
  - Microchip AVR & PIC
  - Și multe altele

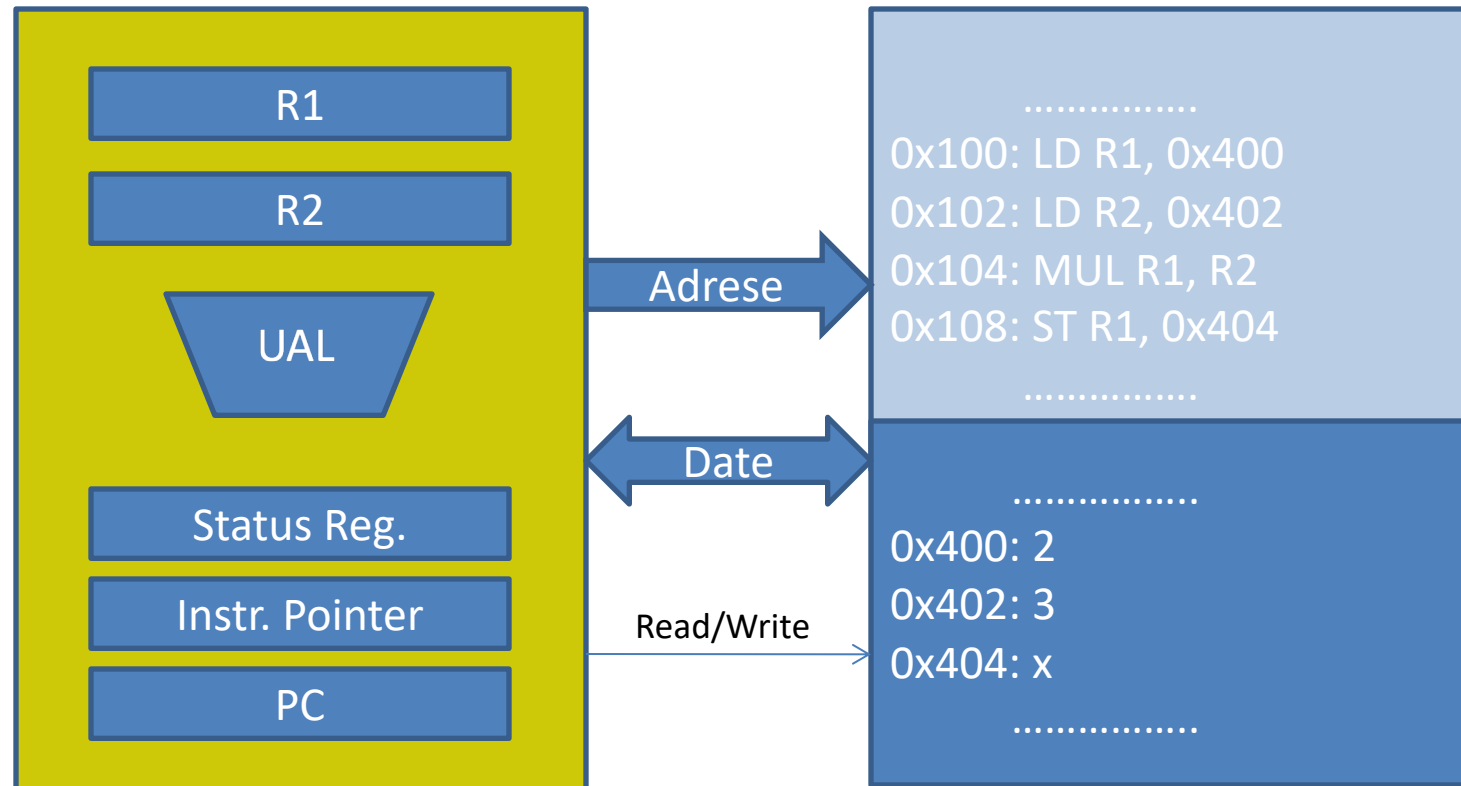


cadence®



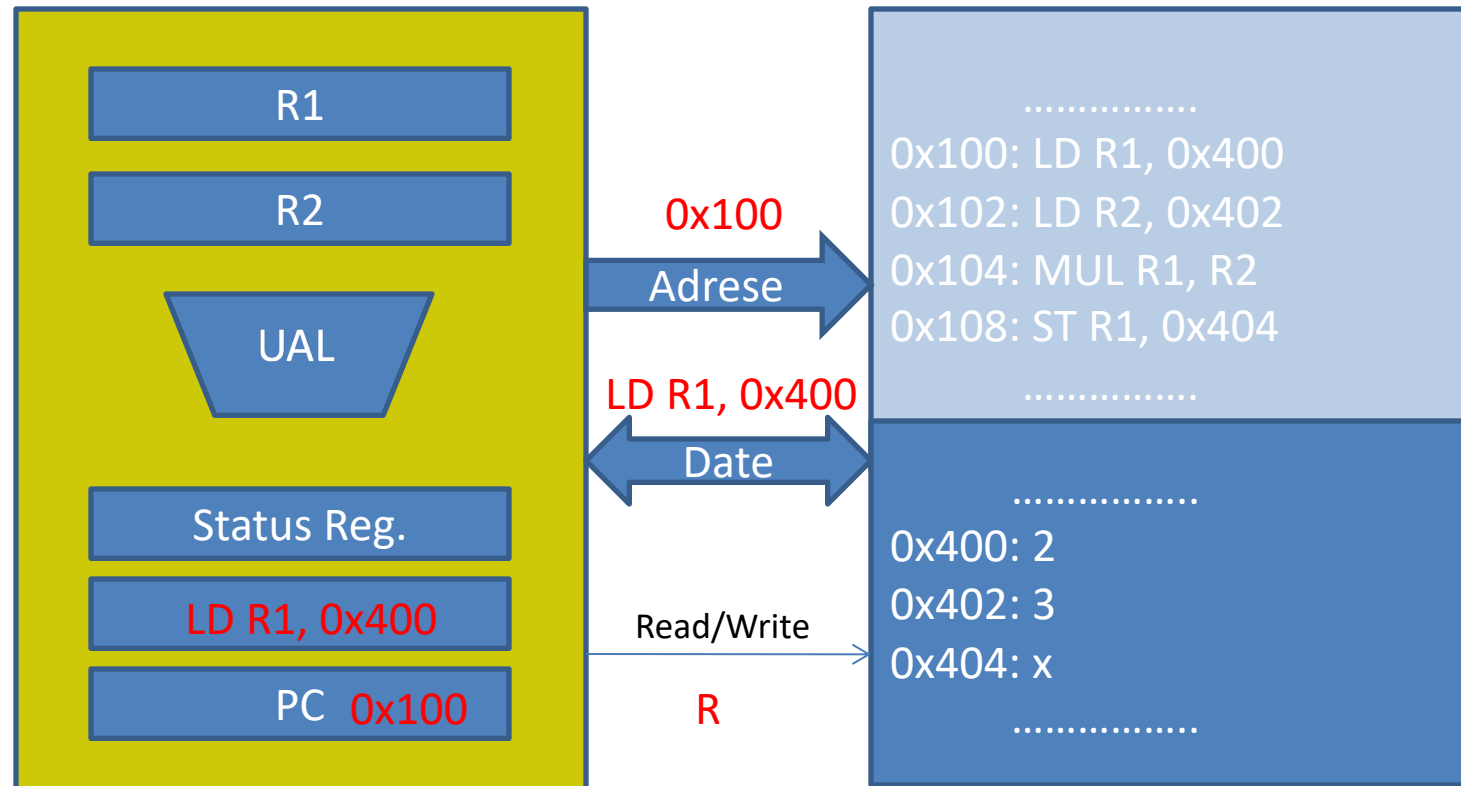


# Arhitectura von Neumann



# Arhitectura von Neumann

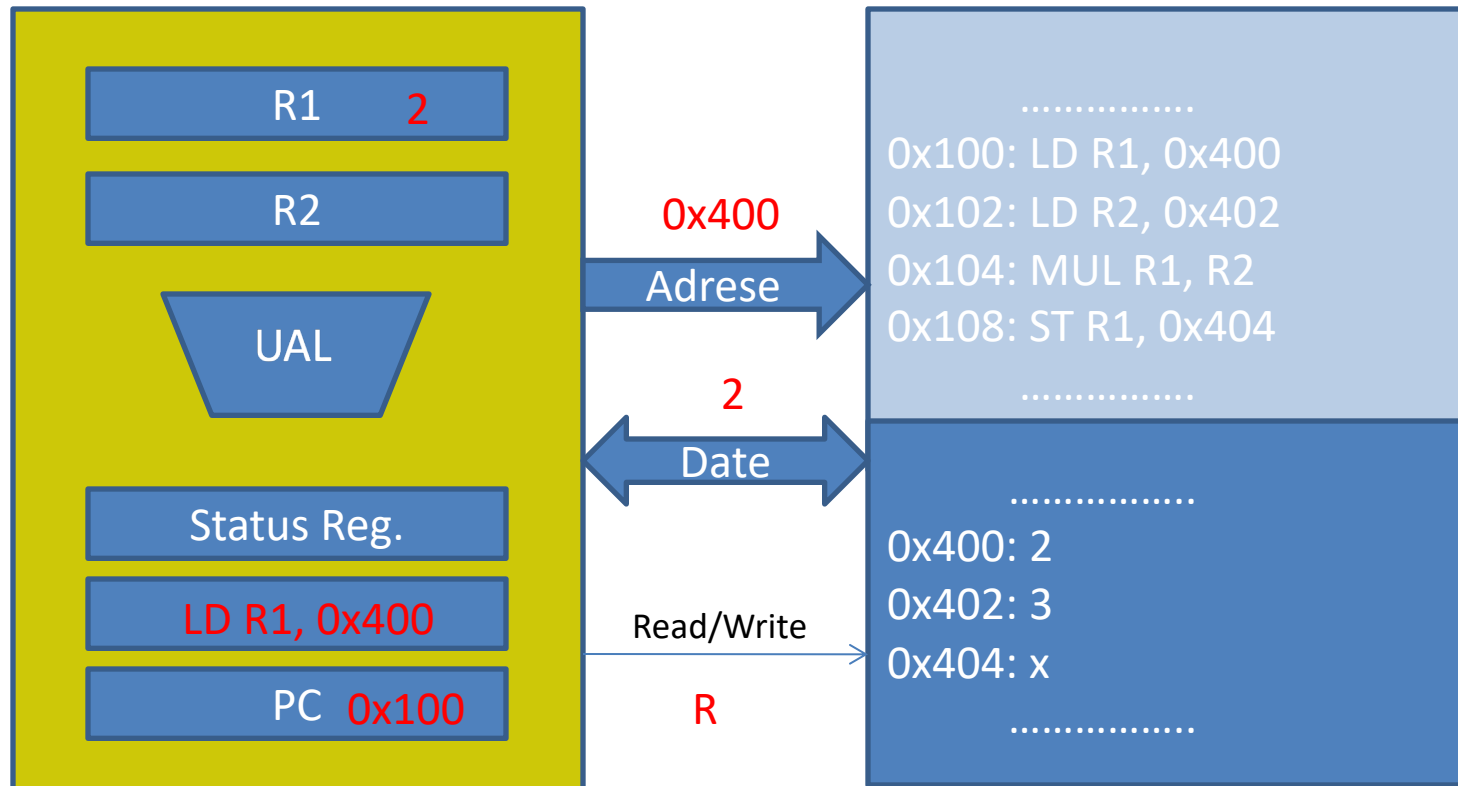
Fetch 0x100



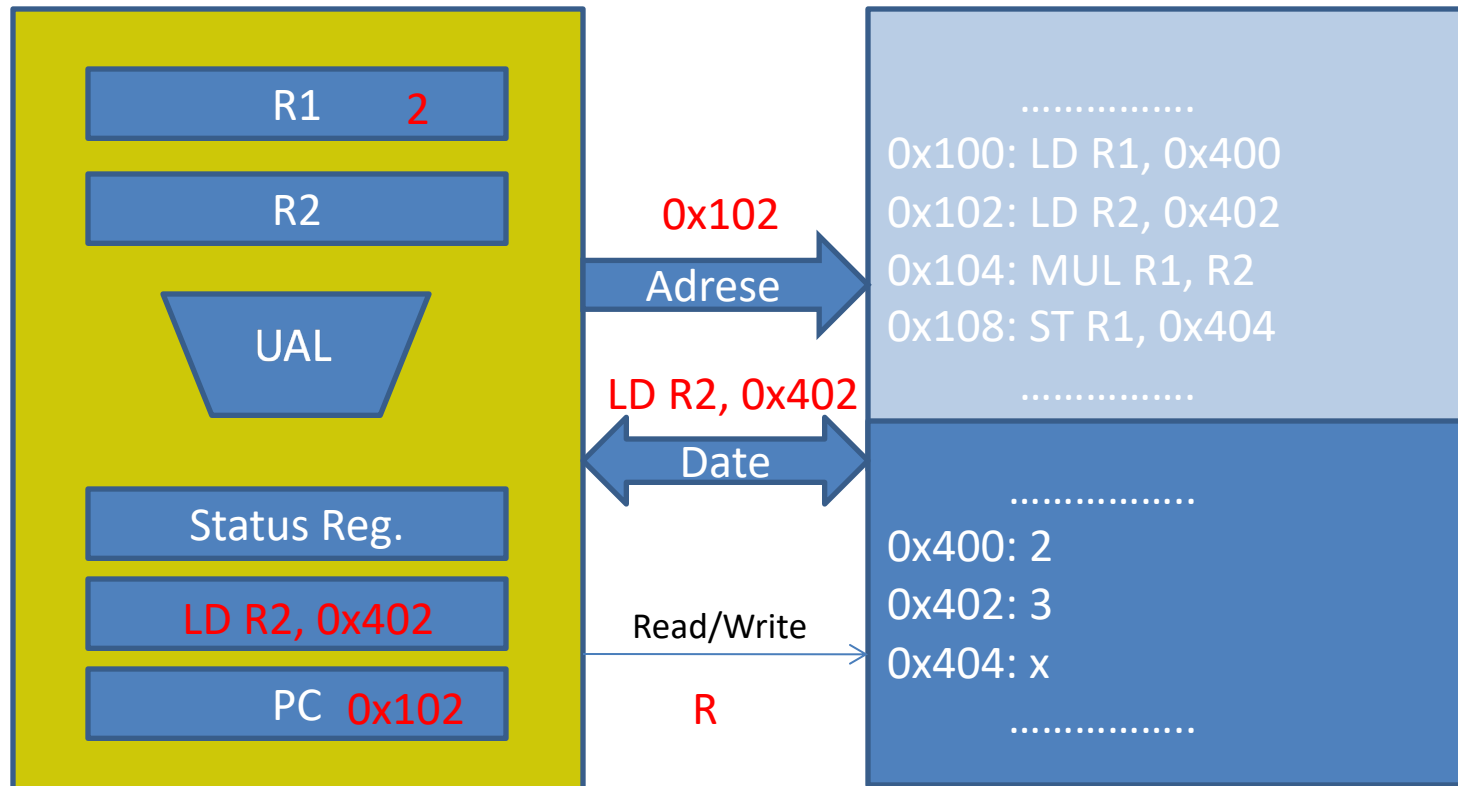


# Arhitectura von Neumann

Execute 0x100

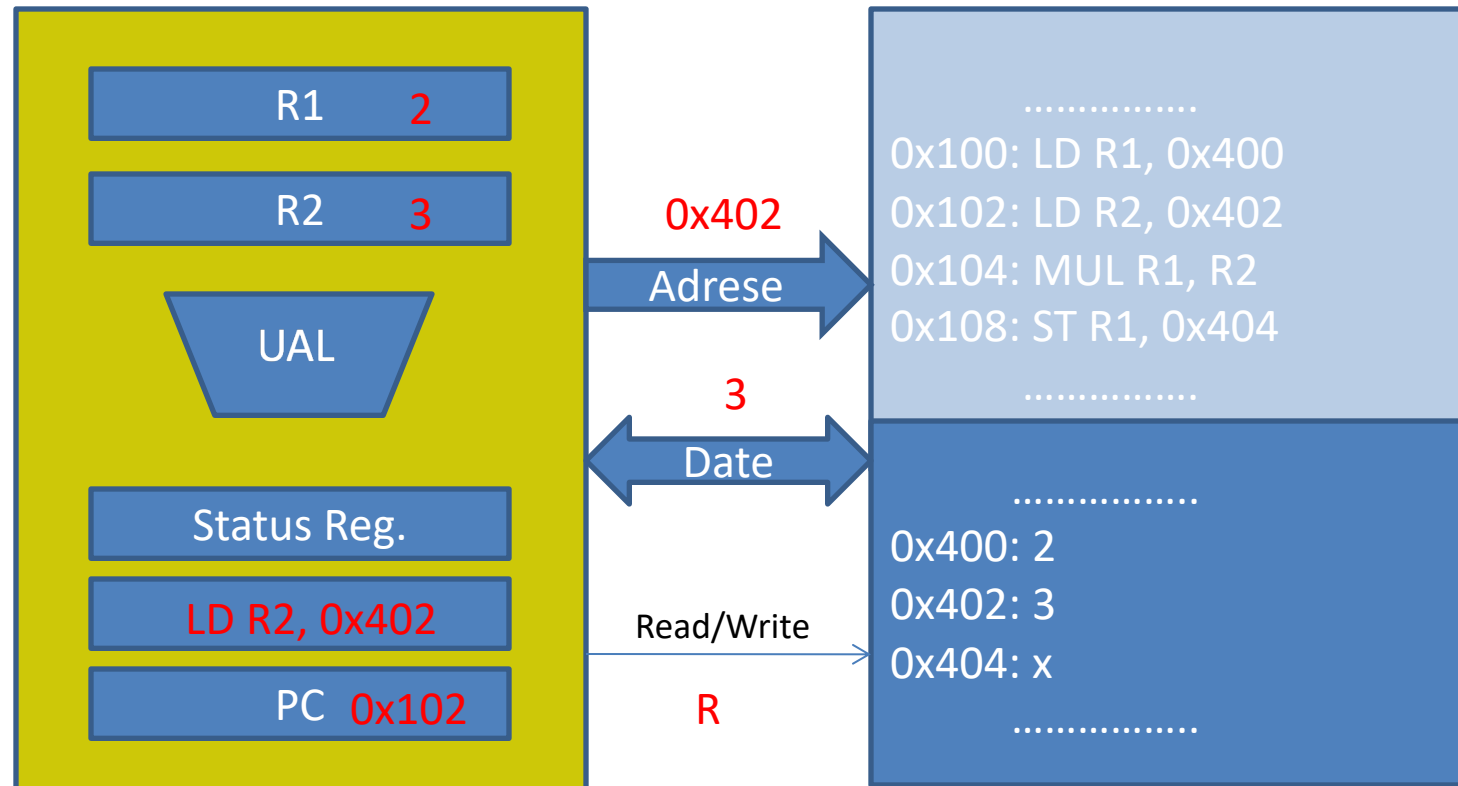


Fetch 0x102



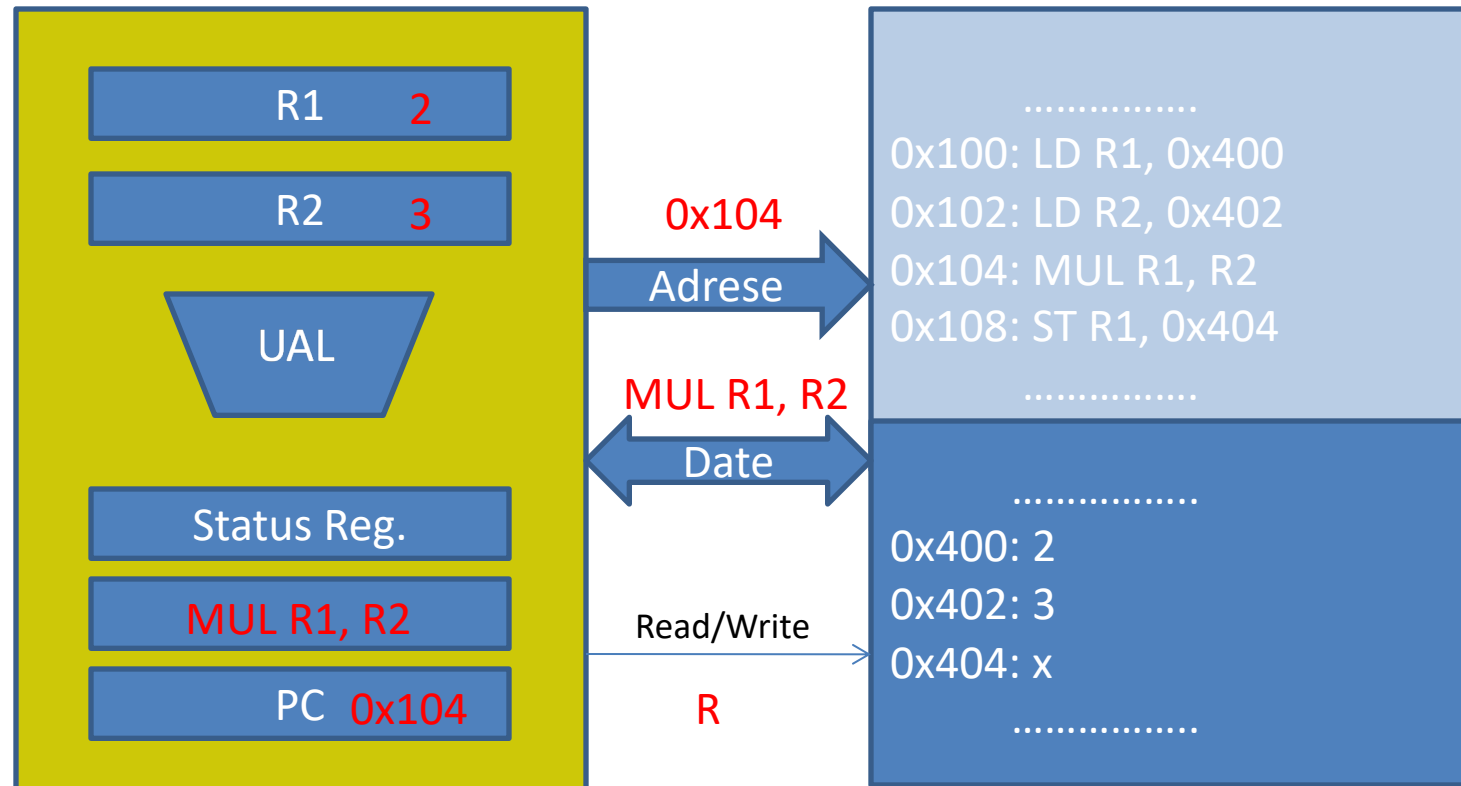
# Arhitectura von Neumann

Execute 0x102



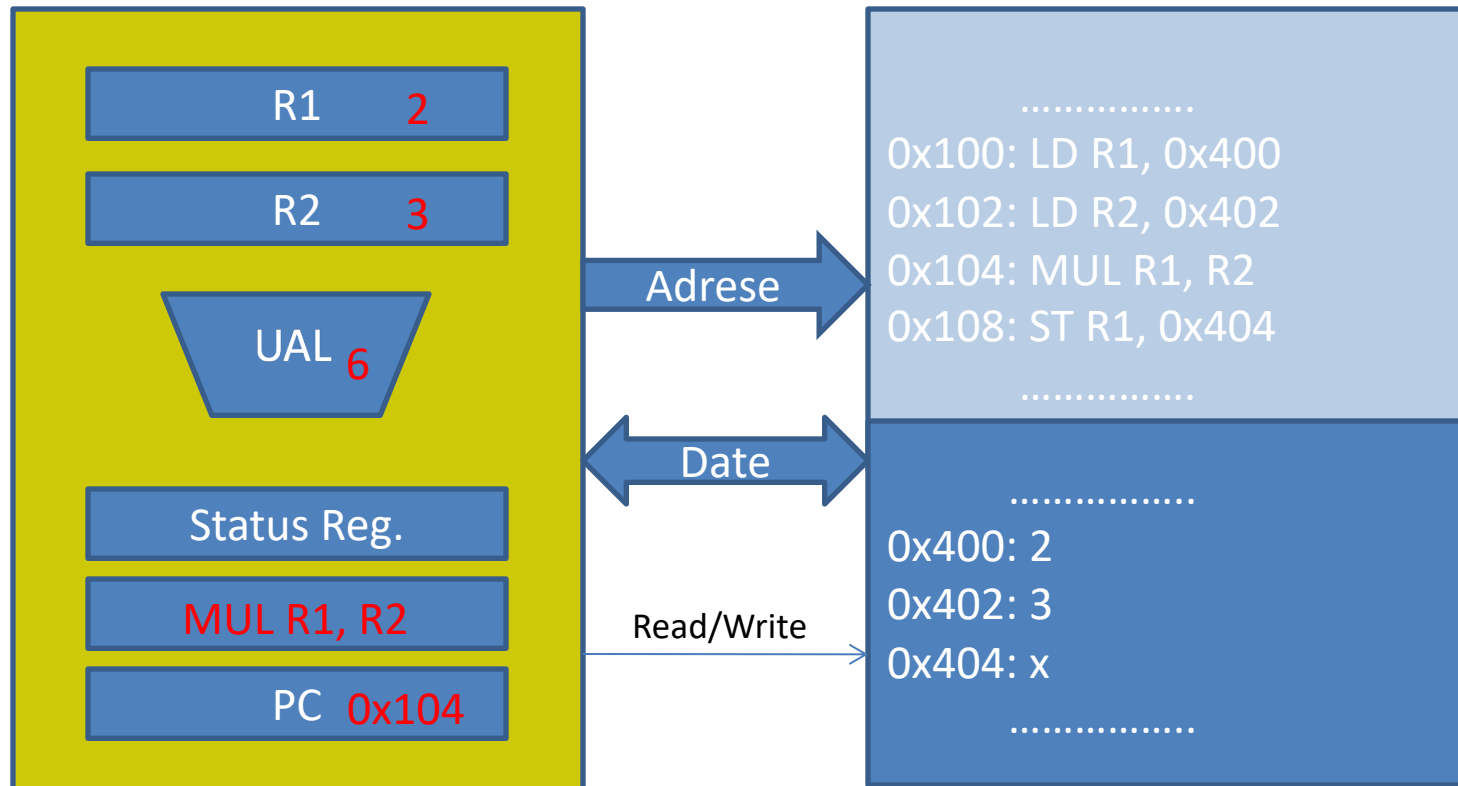
# Arhitectura von Neumann

Fetch 0x104



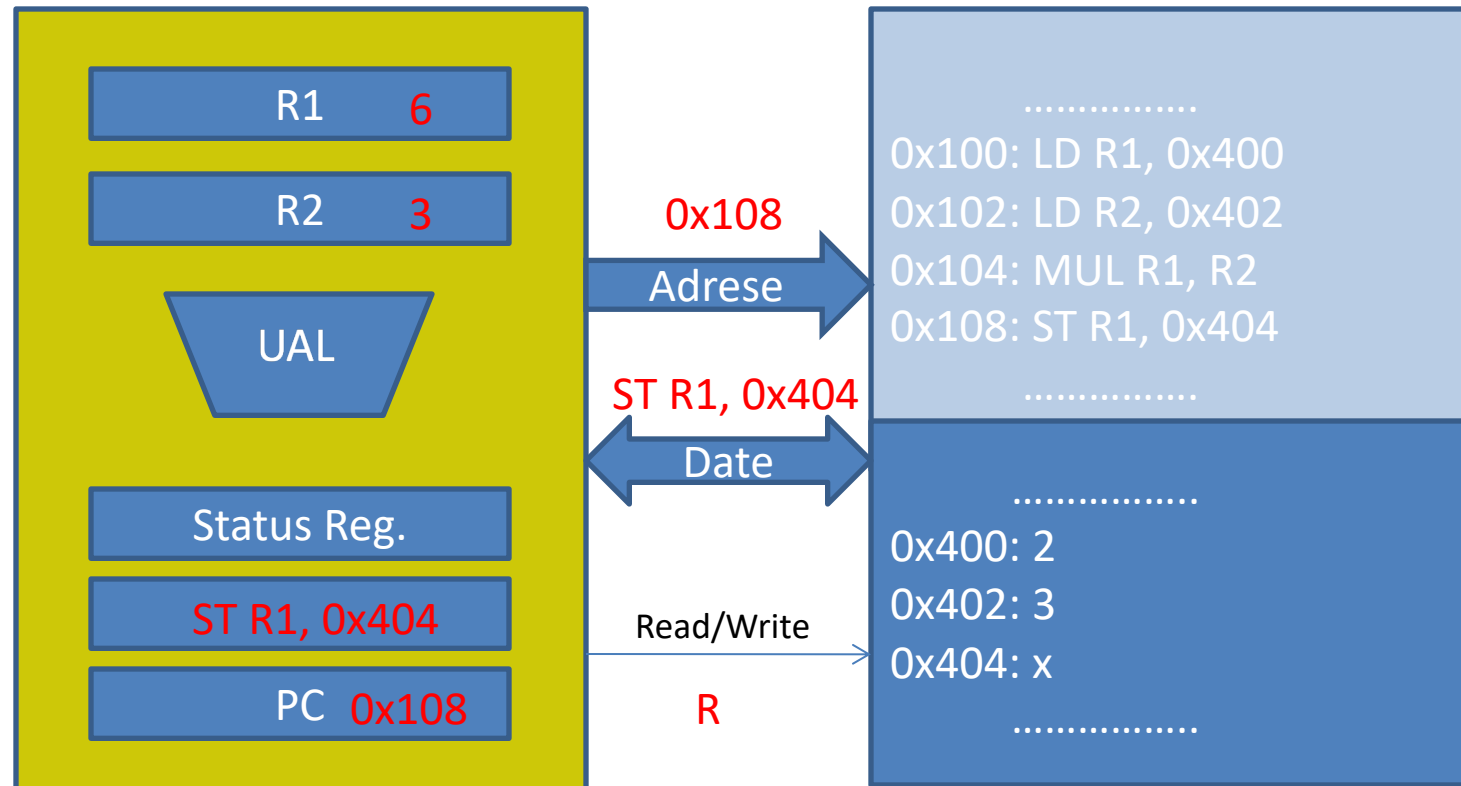
# Arhitectura von Neumann

Execute 0x104



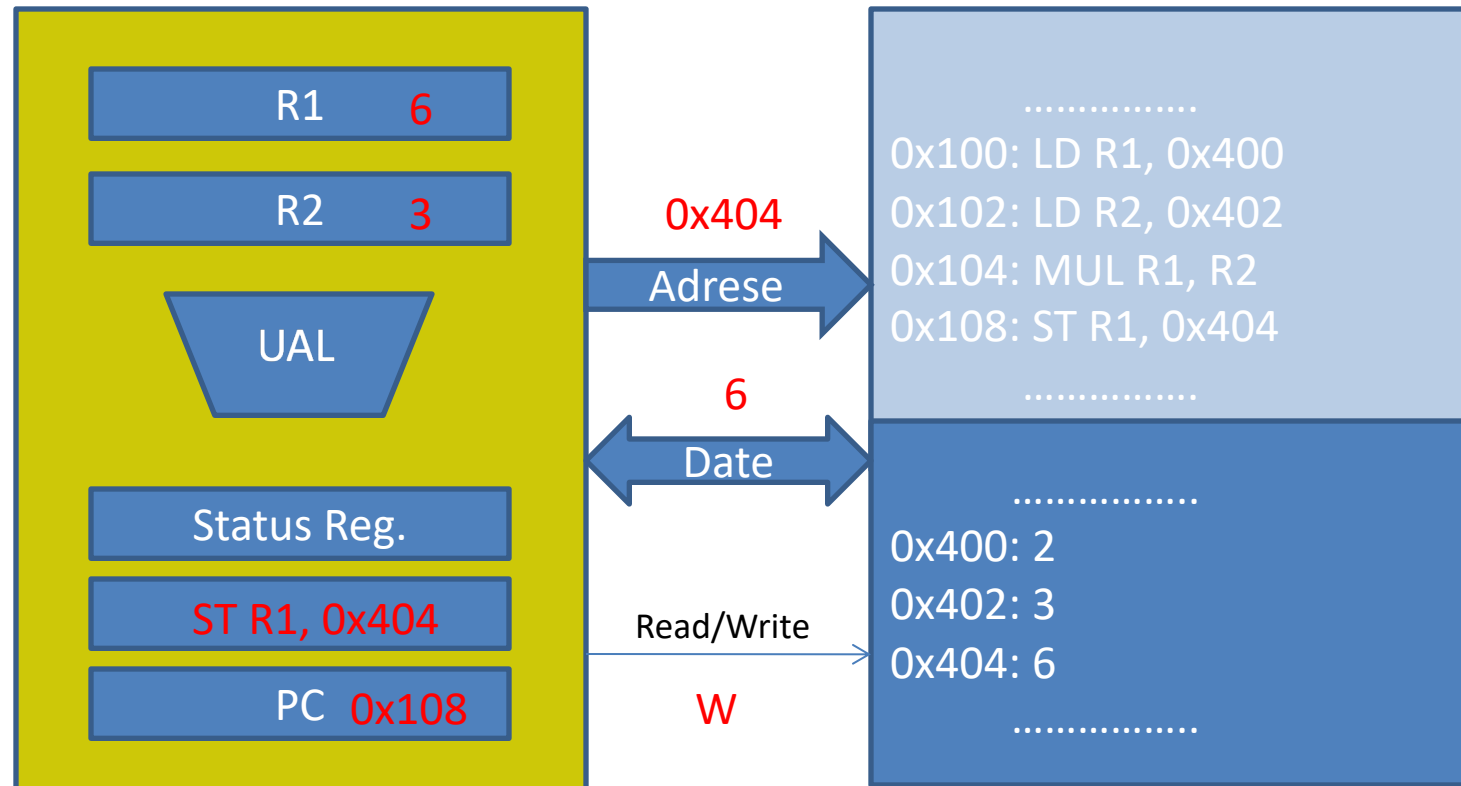
# Arhitectura von Neumann

Fetch 0x108

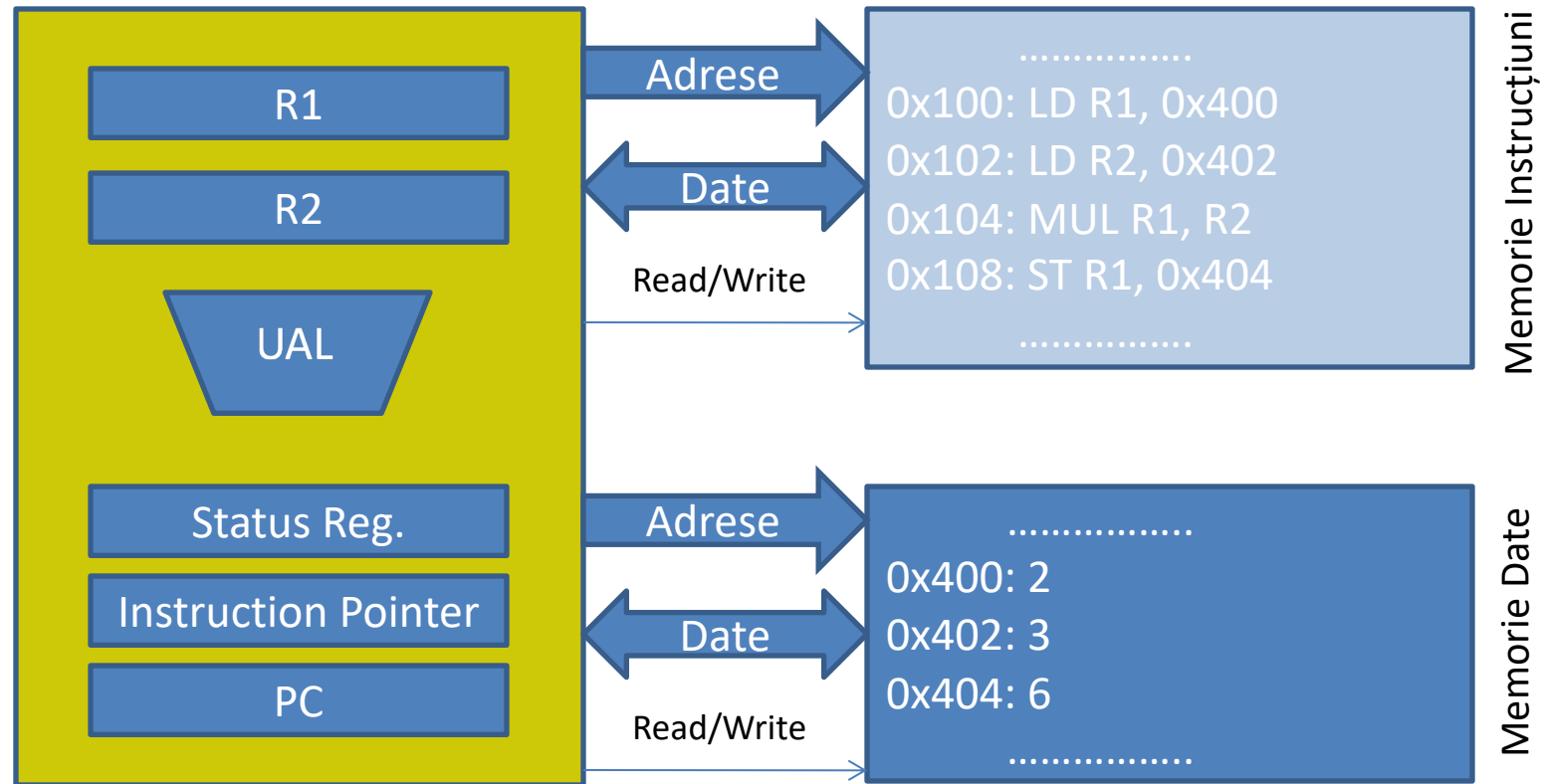


# Arhitectura von Neumann

Execute 0x108



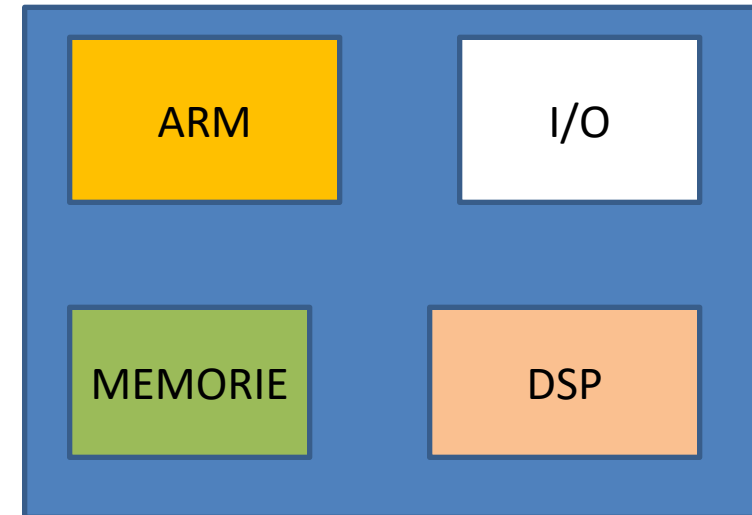
# Arhitectura Harvard





- Memoria de date și memoria de program sunt separate
  - Pot să fie tehnologii diferite (Flash, RAM, EPROM)
- Poate să facă doi cicli de fetch simultan (unul din memoria de date și altul din memoria de program)
- Memoria de program nu poate fi modificată
  - Nu suportă cod care să se auto-modifice
- Majoritatea DSP-urilor au arhitectura Harvard
  - Lațime de bandă mai mare
- Design mai complicat ca von Neumann
  - Hardware în plus pentru cele două magistrale

- Procesoarele ARM sunt o familie de arhitecturi RISC ce împărtășesc aceleași principii de design și un set comun de instrucțiuni.
- ARM este nucleul unui SoC (System on Chip)
- ARM nu produce procesoarele ci vinde licența altor companii care integrează procesorul în sisteme proprii.



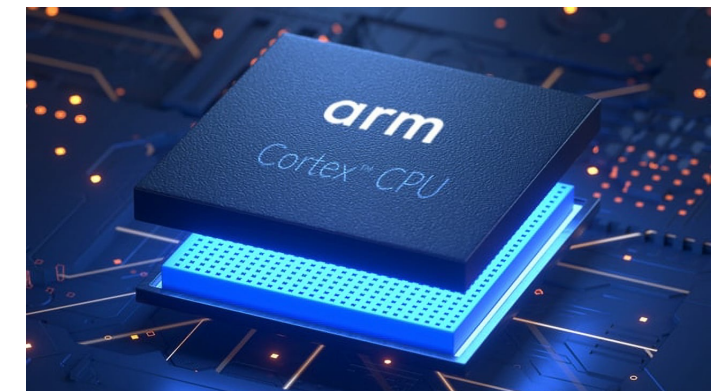
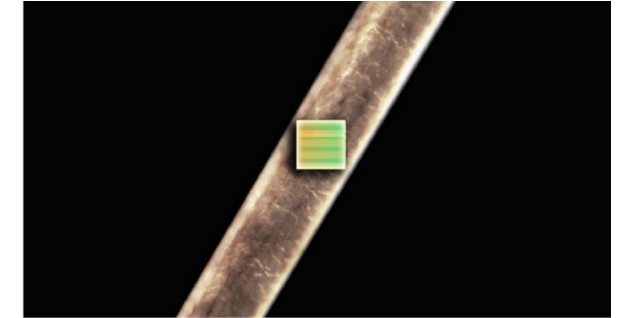
- 1978 – Acorn Computers Ltd. (Cambridge UK)
- 1985 – ARM1 (**A**corn **R**ISC **M**achine)
- 1986 – ARM2
  - Procesor pe 32 de biți
  - Fără memorie cache
  - Cel mai simplu procesor pe 32 de biți din lume (doar 30.000 de tranzistoare)
  - Performanțe superioare unui Intel 80286 contemporan
- 1991 – ARM6 (**A**dvanced **R**ISC **M**achine)
  - Consorțiu Apple – Acorn
  - Primul PDA Apple (Newton)
  - 35.000 tranzistoare



- 1995 – StrongARM
  - Produs DEC sub licență ARM
  - Îmbunătățiri substanțiale de viteză și consum (1W la 233MHz)
- 1997 – Intel StrongARM
  - Achiziționat de la DEC
  - Menit să înlocuiască i860 și i960
- 2001 – ARM7TDMI
  - Cel mai popular procesor ARM
  - Încorporat în iPod, Game Boy Advance, Nintendo DS
- 2002 – 1 miliard de cipuri ARM vândute global
- 2003 – ARM11
  - Nokia N95, iPhone
- 2006 – Marvell Xscale
  - Blackberry
- 2008 – 10 miliarde de cipuri vândute global



- 2009 – Cortex M0
  - arhitectură pe 32 biți
  - Folosit la scară largă în Embedded
  - 15 licențe în primele 9 luni
- 2015 – 1 miliard de cipuri vândute în fiecare lună
- 2015 – OpenFog Consortium
  - Cisco Systems, Dell, Intel, Microsoft și Princeton University
  - Fog computing & wearables
- 2016 – Achiziționat de SoftBank

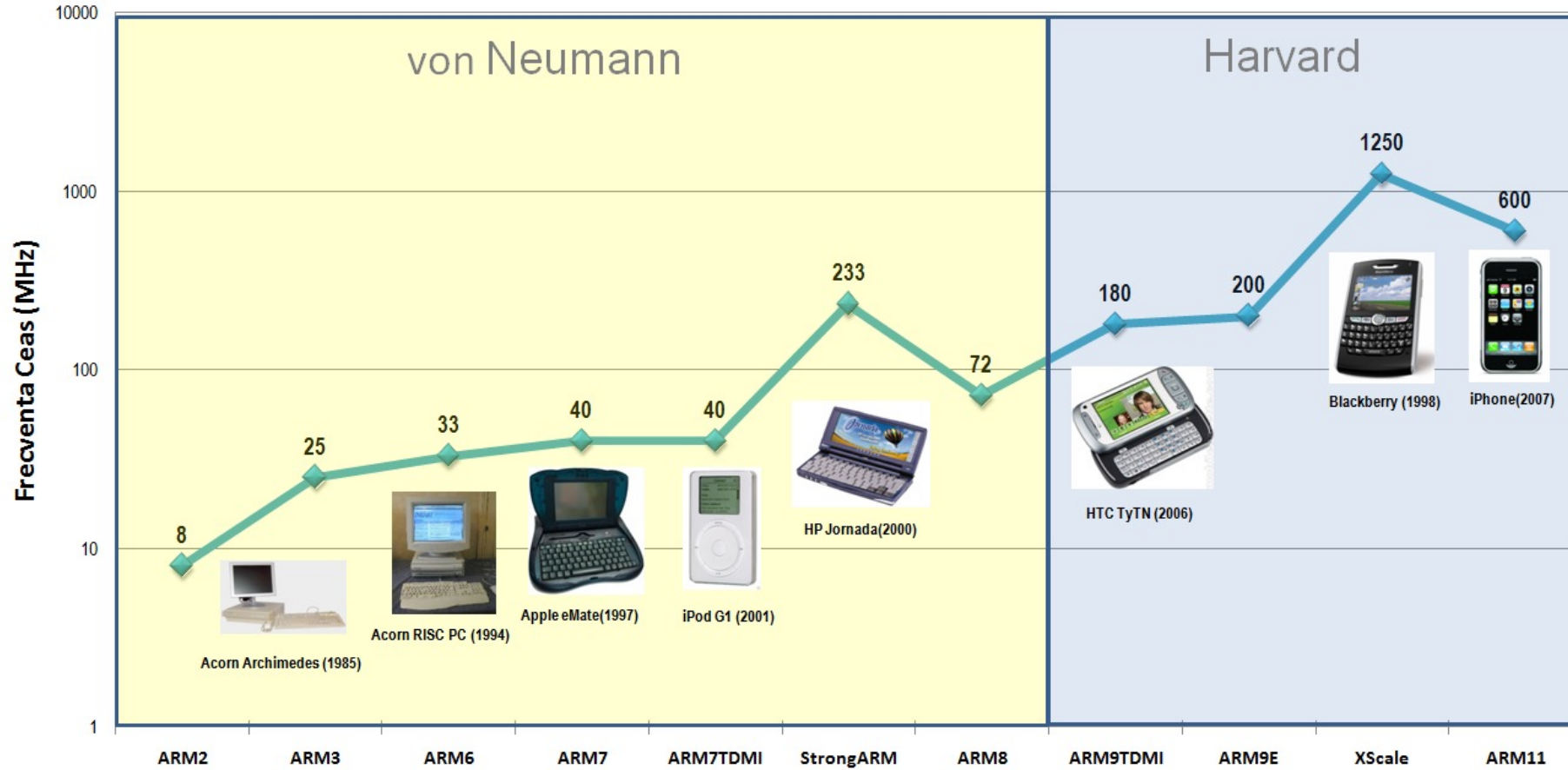


# ARM - Naming Convention

- ARMxyzTDMIEJFS
  - x: series
  - y: MMU
  - z: cache
  - T: Thumb
  - D: debugger
  - M: Multiplier
  - I: EmbeddedICE (built-in debugger hardware)
  - E: Enhanced instruction
  - J: Jazelle (JVM)
  - F: Floating-point
  - S: Synthesizable version (source code version for EDA tools)

- ARMv1-ARMv3: pierdute în negura vremurilor
- ARMv4T: ARM7TDMI – first Thumb processor
- ARMv5TEJ(+VFPv2): ARM926EJ-S
- ARMv6K(+VFPv2): ARM1136JF-S, ARM1176JFZ-S,  
ARM11MPCore – first Multiprocessing Core
- ARMv7-A+VFPv3 Cortex-A8
- ARMv7-A+MPE+VFPv3: Cortex-A5, Cortex-A9
- ARMv7-A+MPE+VE+LPAAE+VFPv4 Cortex-A15
  
- ARMv7-R : Cortex-R4, Cortex-R5
- ARMv6-M Cortex-M0
- ARMv7-M: Cortex-M3, Cortex-M4

# ARM – Evoluție





- Nucleul ARM
  - tablete
  - telefoane mobile (nu numai smartphone)
  - console de jocuri
  - wearables
- Costul redus de fabricație și consumul mic de energie electrică au făcut din ARM cel mai utilizat procesor din lume
  - peste 10 miliarde de procesoare vândute până în 2008
  - 2011: 5 miliarde de procesoare anual
- Procesoarele variază în funcție de versiune și:
  - Memoria cache
  - Lățimea magistralei
  - Frecvența de ceas

# Ecosistemul ARM



**ATAP Partners**

**Tools Partners**

**RTOS Partners**

**Software Partners**

**ARM in Partnership**

The image displays a large collection of logos for various companies and organizations that are part of the ARM ecosystem. The logos are arranged in a grid-like fashion, with a central 'ARM in Partnership' logo. The logos are categorized into four main groups: ATAP Partners, Tools Partners, RTOS Partners, and Software Partners. The central logo features the ARM logo and the text 'ARM in Partnership'.

# Aplicații ARM



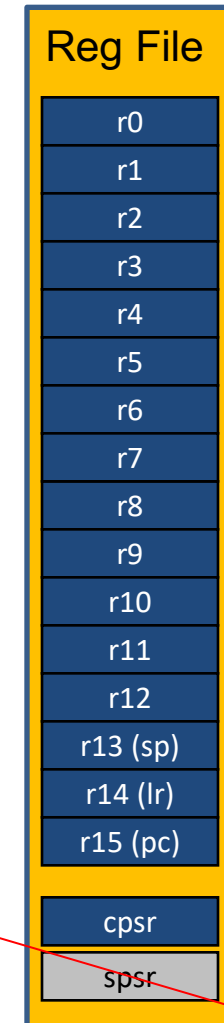
- ARM este o arhitectură pe 32 de biți.
  - Versiuni diferite pot utiliza arhitecturi diferite
    - ARM 7: von Neumann
    - ARM 9: Harvard
- Majoritatea chipurilor ARM implementează două seturi de instrucțiuni
  - ARM Instruction Set (32 biti)
  - Thumb Instruction Set (16 biti)
- Jazelle core – procesorul execută bytecode Java
  - Majoritatea instrucțiunilor executate direct în hardware
  - Instrucțiunile complexe sunt executate în software
  - Compromis între complexitate hardware / viteză de execuție

- ARM are șapte moduri de operare:
  - **User** : mod neprivilegiat; cele mai multe task-uri rulează aici
  - **FIQ** : declanșat la producerea unei întreruperi de prioritate mare (fast interrupt)
  - **IRQ** : declanșat la producerea unei întreruperi de prioritate normală (low interrupt)
  - **Supervisor** : declanșare la RESET și la întreruperi software
  - **Abort** : la acces nepermis la memorie
  - **Undef** : pentru instrucțiuni nedefinite
  - **System** : mod privilegiat; folosește aceleași registre ca User Mode

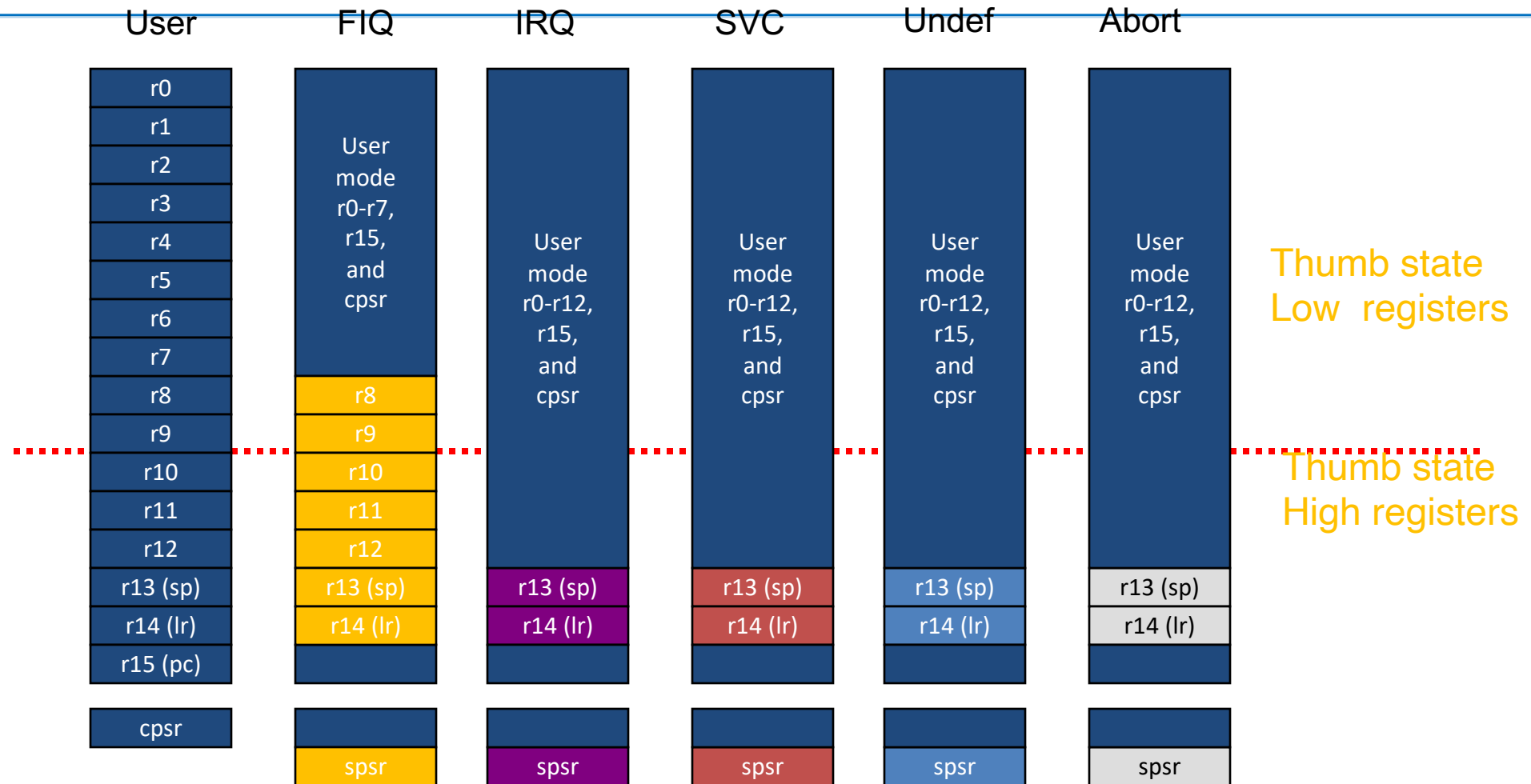
# ARM – User Mode

Procesorul ARM are 17 registre active in User Mode

- 16 registre de date (*r0-r15*)
- 1 registru *processor status*
- *r13-r15* au și alte funcții
  - ✓ *r13* stack pointer (*sp*)
  - ✓ *r14* link register (*lr*)
  - ✓ *r15* program counter (*pc*)
- *CPSR* – *Current Program Status Register*
- *SPSR* - *Saved Program Status Register*
  - ✓ salvează o copie a *CPSR* la producerea unei excepții
  - ✓ nu este disponibil în *User Mode*



# ARM – Toate registrele



Nota: System mode folosește același set de registre ca și User mode

# ARM - Registre

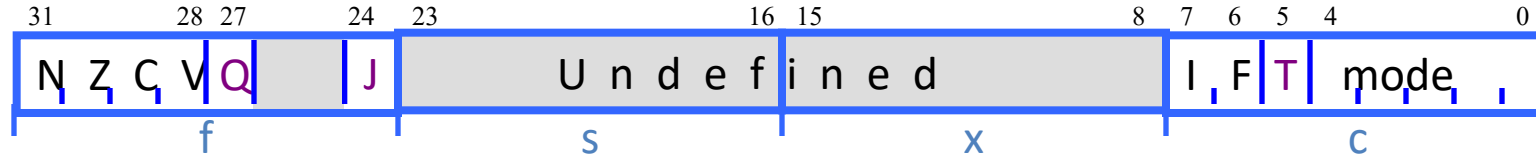
- ARM are 37 de registre de 32 de biti.
  - 1 program counter
  - 1 current program status register (CPSR)
  - 5 saved program status register (SPSR)
  - 30 registre general purpose
- Modul curent de lucru determina care banc de memorie este accesibil in acel moment. Indiferent de privilegii, fiecare mod poate accesa
  - registrele **r0-r12**
  - **r13** (stack pointer, **sp**) si **r14** (link register, **lr**)
  - registrul program counter, **r15** (**pc**)
  - current program status register, **cpsr**

Modurile privilegiate (mai putin System) pot sa acceseze si

- un registru **spsr** (saved program status register)



# ARM - CPSR



- Condition Flags
  - N = rezultat **N**egativ UAL
  - Z = rezultat **Z**ero UAL
  - C = **C**arry Flag
  - V = **oV**erflow flag
- Sticky Overflow **Q** – flag
  - Nu poate fi sters de alte operatii; decat printr-o instructiune dedicata.
- Bitul **J**
  - J = 1: Procesorul in starea Jazelle
- Interrupt Disable.
  - I = 1: IRQ Disable.
  - F = 1: FIQ Disable.
- Bitul **T**
  - T = 0: Procesor in starea ARM
  - T = 1: Procesor in starea Thumb
- Biti **Mode**
  - Specifica modul procesorului

# ARM – Program Counter

- Cand procesorul este in starea ARM:
  - Toate instructiunile au 32 de biti lungime
  - Instructiunile sunt aliniate la nivel de cuvant
  - Valoarea PC este stocata in bitii [31:2] iar bitii [1:0] sunt nedefiniti.
- Cand procesorul este in starea Thumb:
  - Instructiunile au 16 biti lungime
  - Toate instructiunile sunt aliniate la nivel de jumatate de cuvant
  - Valoarea PC e stocata in bitii [31:1] cu bitul [0] nedefinit (instructiunile nu pot fi aliniate la nivel de octet).
- Cand procesorul este in starea Jazelle:
  - Toate instructiunile au 8 biti lungime
  - Procesorul citeste patru instructiuni simultan (citeste un cuvant de memorie)

# ARM – Tratarea Intreruperilor

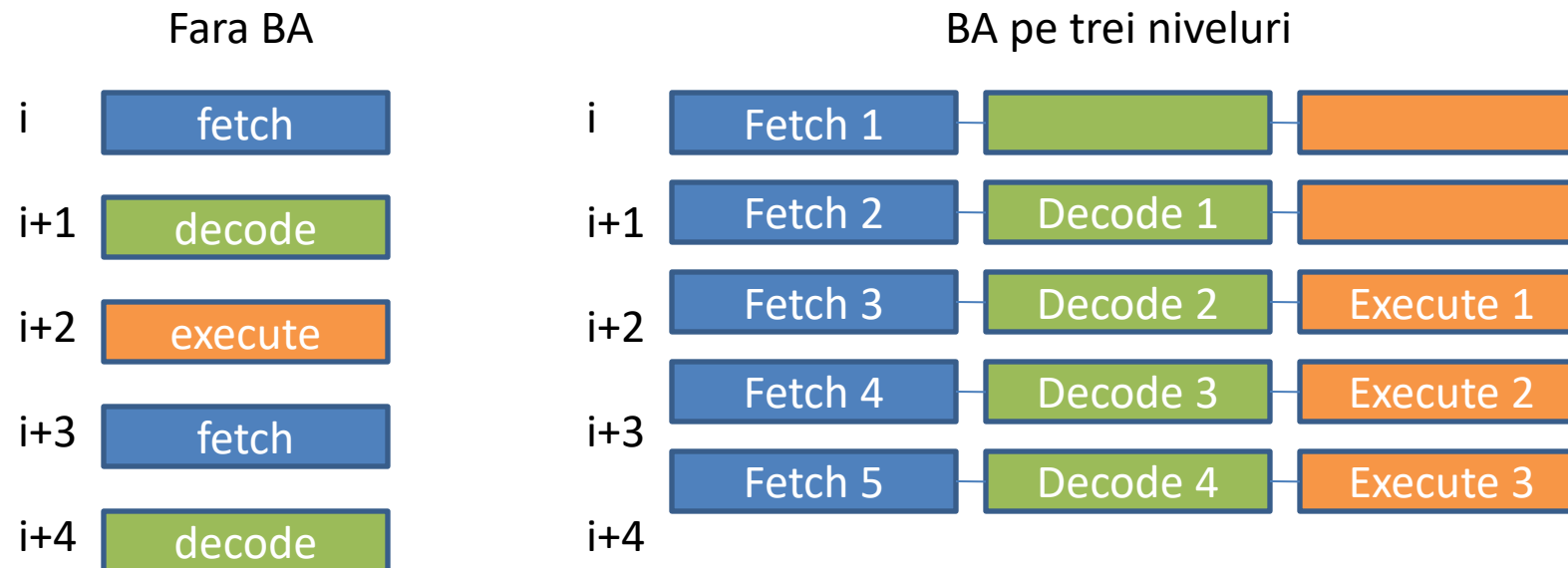
- La aparitia unei exceptii:
  - Copiaza CPSR in registrul SPSR\_<mod>
  - Seteaza bitii din CPSR
    - Intoarcere la starea ARM
    - Intrare in modul intrerupere
    - Opreste alte intreruperi (dupa caz)
  - Salveaza adresa de revenire in LR\_<mod>
  - Seteaza PC la adresa vectorului curent
- La revenirea dintr-o intrerupere, un interrupt-handler:
  - Reface CPSR din SPSR\_<mod>
  - Reface PC din LR\_<mode>

	⋮
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

# ARM – Banda de Asamblare

- Banda de asamblare este folosita pentru a mari numarul de operatii executate per ciclu



# ARM – Banda de Asamblare

- Banda de asamblare ARM7 are 3 niveluri



- Banda de asamblare ARM9 are 5 niveluri



# ARM – Executie Conditionata

- Executia conditionala este realizata prin adaugarea de instructiuni care testeaza bitii de status dupa operatia curenta.
  - Reduce densitatea codului prin evitarea salturilor conditionale.

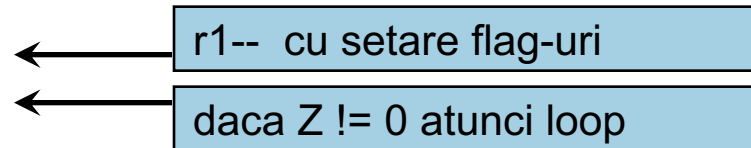
```
CMP    r3,#0
BEQ    skip
ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
ADDNE  r0,r1,r2
```

- In general, instructiunile de procesare a datelor nu afecteaza bitii de status. Acestia pot fi setati optional folosind sufixul “S”.

loop

```
...
SUBS  r1,r1,#1
BNE  loop
```



# ARM – Sufixe Conditionale

- Toate sufixele conditionale posibile:
  - AL este optiunea default; nu trebuie specificat

Sufix	Descriere	Flag-uri testate
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

# ARM - Exemple

- Secventa de instructiuni conditionale

```
if (a==0) func(1);  
    CMP        r0,#0  
    MOVEQ     r0,#1  
    BLEQ     func
```

- Seteaza flag conditie, apoi executa

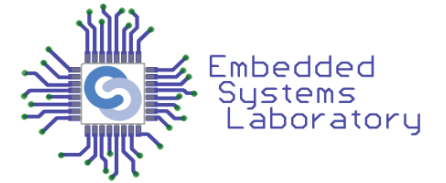
```
if (a==0) x=0;  
if (a>0)  x=1;  
    CMP        r0,#0  
    MOVEQ     r1,#0  
    MOVGT     r1,#1
```

- Instructiuni de comparatie conditionale

```
if (a==4 || a==10) x=0;  
    CMP        r0,#4  
    CMPNE     r0,#10  
    MOVEQ     r1,#0
```



# ARM – Instructiuni de procesare date



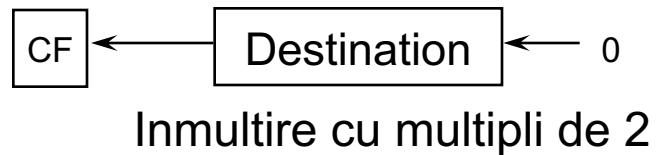
- Sunt de 4 tipuri :
  - Aritmetice:            **ADD**        **ADC**        **SUB**        **SBC**        **RSB**        **RSC**
  - Logice:                **AND**        **ORR**        **EOR**        **BIC**
  - Comparatii:         **CMP**        **CMN**        **TST**        **TEQ**
  - Manipulare date:   **MOV**        **MVN**
- Functioneaza doar cu registre generale, nu si cu locatii de memorie.
- Sintaxa:

**<Operatie>{<cond>}{S} Rd, Rn, Operand2**

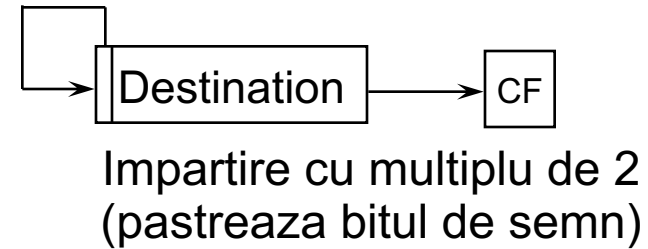
  - Comparatiile seteaza flag-urile – nu e specificat Rd
  - MOV, MVN nu specifica Rn
- Al doilea operand este trimis UAL prin Barrel Shifter.

# ARM – Barrel Shifter

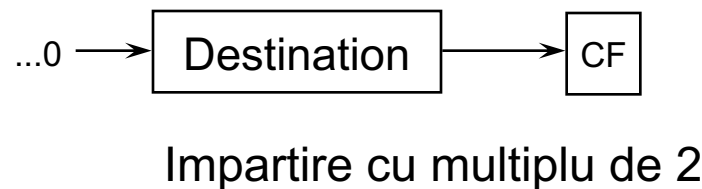
## LSL : Logical Left Shift



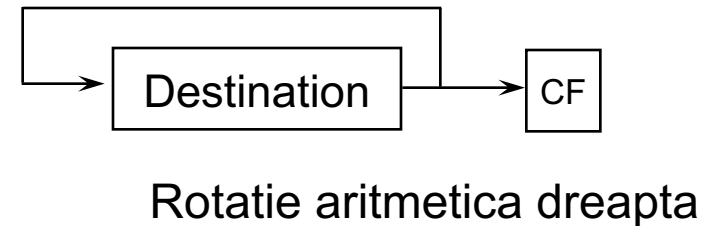
## ASR: Arithmetic Right Shift



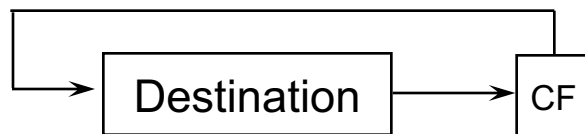
## LSR : Logical Shift Right



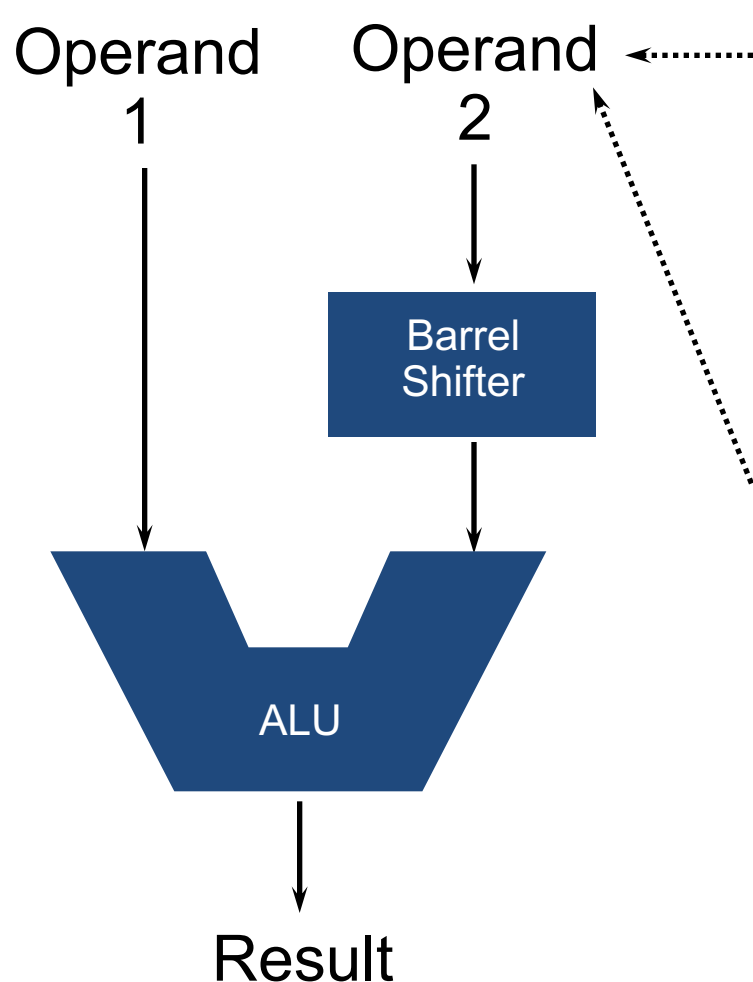
## ROR: Rotate Right



## RRX: Rotate Right Extended



# ARM – Barrel Shifter



## Registru, optional cu operatie shiftare

- Numarul de rotatii:
  - intreg fara semn pe 5 biti
  - Ultimul octet al altui registru
- Folosit pentru inmultirea cu o constanta

## Valoare imediata

- Constanta pe 8 biti
  - Rotita dreapta

## Exemple:

```
ADD    r0, r1, r2
ADD    r0, r1, r2, LSL#7
ADD    r0, r1, r2, LSL r3
ADD    r0, r1, #0x4E
```

# Exemple de cod

if (z==1) R1=R2+(R3\*4)

se compilează în

EQADDS R1,R2,R3, LSL #2

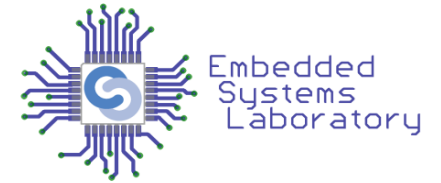
**O singură instrucțiune!**

# Exemple de cod

```
do
{
    if(i > j) // When i>j we do that
        i -= j;
    else if(i < j) // When i<j we do that
        j -= i;
    else ;      // When i==j we do nothing
} while(i != j); // Leave the loop when i==j
```

```
loop: CMP    Ri, Rj          ; set condition "NE" if (i != j),
                           ; "GT" if (i > j),
                           ; or "LT" if (i < j)
      SUBGT  Ri, Ri, Rj      ; if "GT" (Greater Than), i = i-j;
      SUBLT  Rj, Rj, Ri      ; if "LT" (Less Than), j = j-i;
      BNE   loop            ; if "NE" (Not Equal), then loop
```

# ARM – Lucrul cu Constante



- Asamblorul permite incarcarea de constante pe 32 de biti printr-o pseudo-instructiune speciala:
  - `LDR rd, =const`
- Pot fi doua rezultate:
  - O instructiune `MOV` care incarca valoarea in registru.sau
  - Genereaza un `LDR` cu o adresa indexata prin PC pentru a citi constanta dintr-o zona a codului dedicata constantelor (*literal pool*)
- De exemplu
  - `LDR r0, =0xFF` ==> `MOV r0, #0xFF`
  - `LDR r0, =0x55555555` ==> `LDR r0, [PC, #Imm12]`
    - ...
    - ...  
`DCD 0x55555555`
- Este modalitatea recomandata de a incarca constante in cod

- Sintaxa:

- `MUL`{<cond>}{S} Rd, Rm, Rs
- `MLA`{<cond>}{S} Rd,Rm,Rs,Rn
- `[U|S]MULL`{<cond>}{S} RdLo, RdHi, Rm, Rs
- `[U|S]MLAL`{<cond>}{S} RdLo, RdHi, Rm, Rs

$$Rd = Rm * Rs$$

$$Rd = (Rm * Rs) + Rn$$

$$RdHi,RdLo := Rm*Rs$$

$$RdHi,RdLo := (Rm*Rs)+RdHi,RdLo$$

- Cicli de instructiune

- MUL
  - 2-5 cicli pentru ARM7TDMI
  - 1-3 cicli pentru StrongARM/XScale
  - 2 cicli pentru ARM9E/ARM102xE
- +1 ciclu pentru ARM9TDMI (fata de ARM7TDMI)
- +1 ciclu pentru inmultirea acumulata (MLA)
- +1 ciclu pentru “long”

# ARM – Load / Store

<b>LDR</b>	<b>STR</b>	Word
<b>LDRB</b>	<b>STRB</b>	Byte
<b>LDRH</b>	<b>STRH</b>	Halfword
<b>LDRSB</b>		Signed byte
<b>LDRSH</b>		Signed halfword

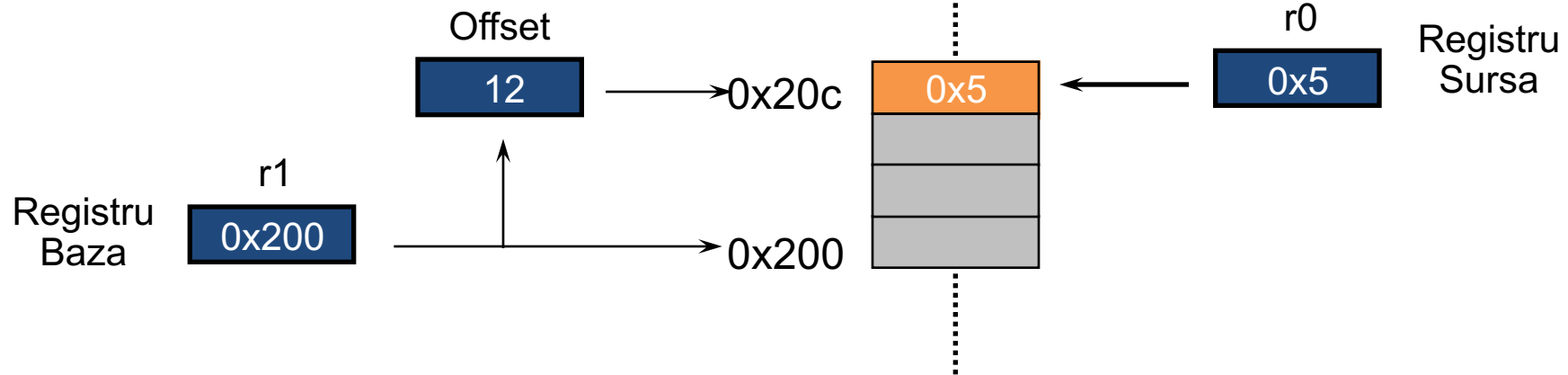
- Memoria trebuie sa suporte toate tipurile de acces.
- Sintaxa:
  - **LDR**{<cond>}{<size>} Rd, <address>
  - **STR**{<cond>}{<size>} Rd, <address>

e.g. **LDREQB r0, [r1, #8]**

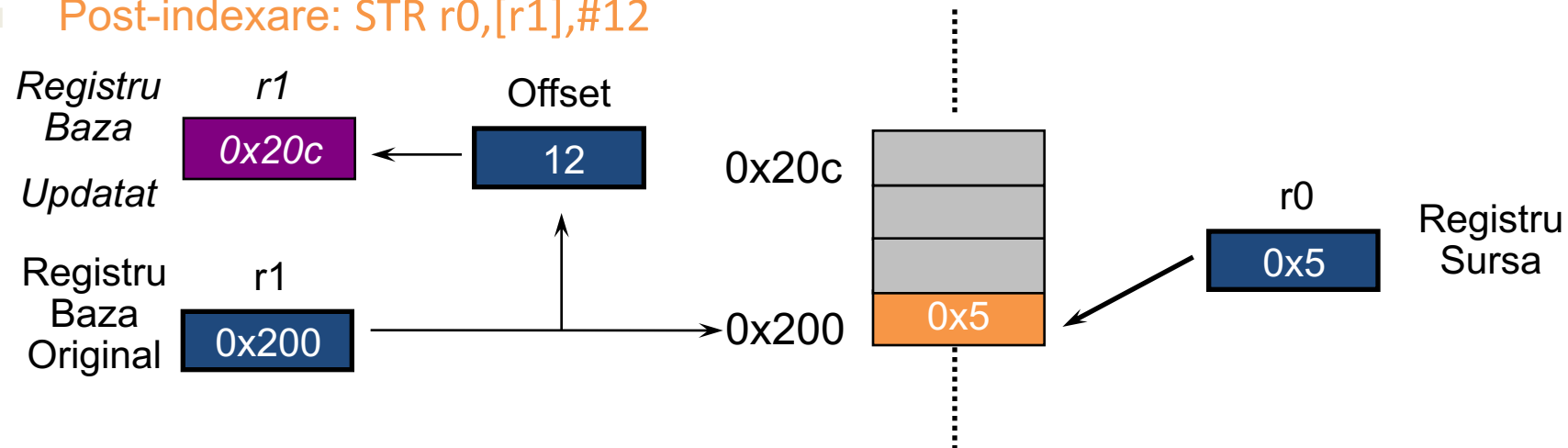


# ARM Load / Store

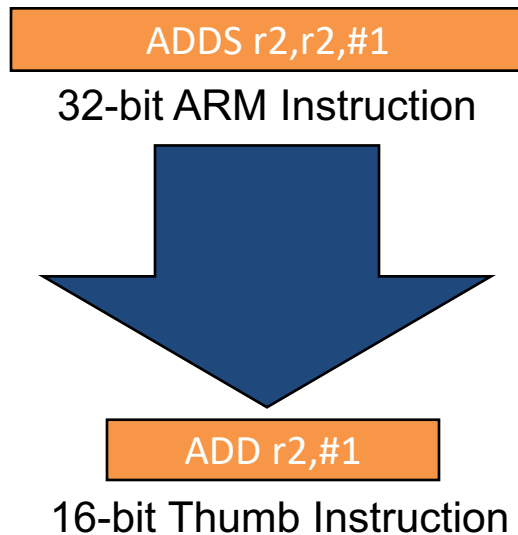
- **Pre-indexare:** `STR r0, [r1, #12]`



- **Post-indexare:** `STR r0, [r1], #12`



- Thumb este un set de instructiuni pe 16 biti
  - Optimizat pentru cod C
  - Performanta marita chiar cu memorie ingusta
  - Subset de functii al setului de instructiuni ARM
- Nucleul procesorului are o stare speciala de executie – Thumb
  - Ciclare intre ARM si Thumb cu instructiunea **BX**

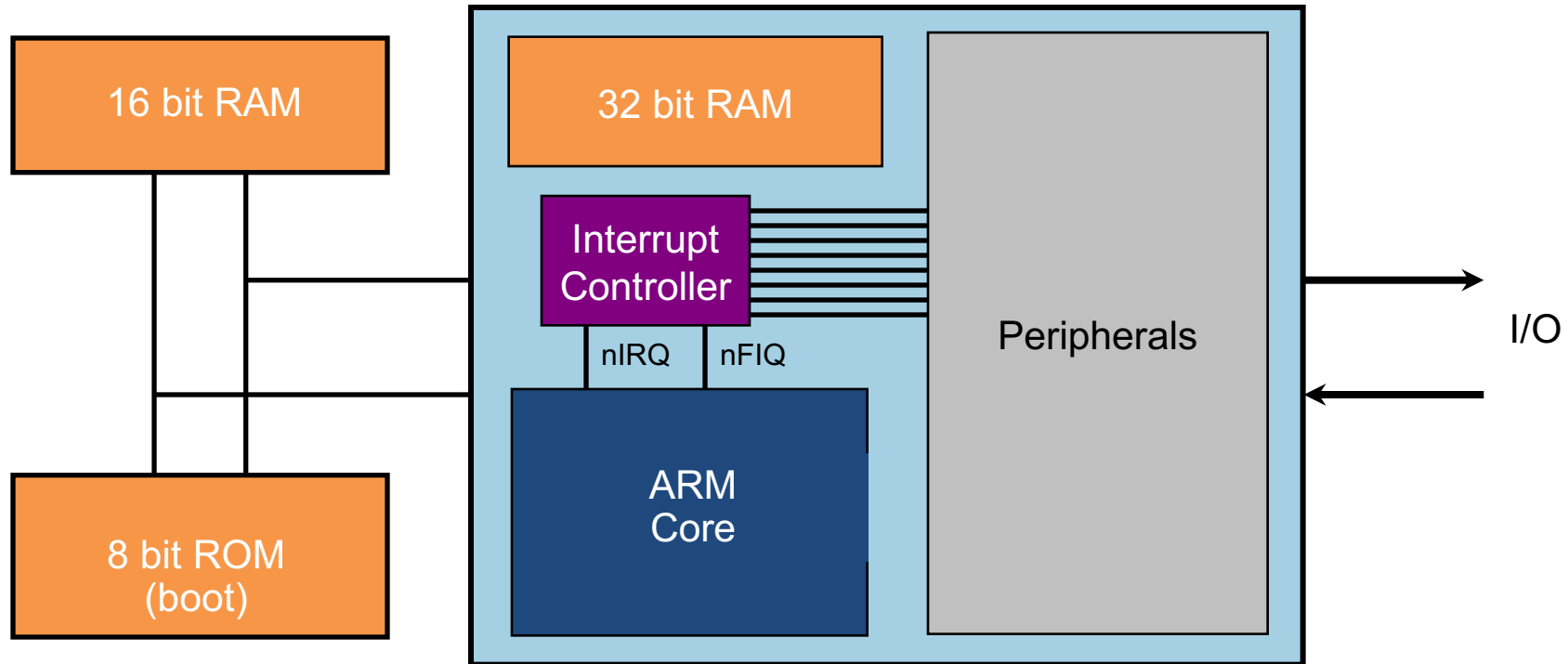


De ce Thumb?

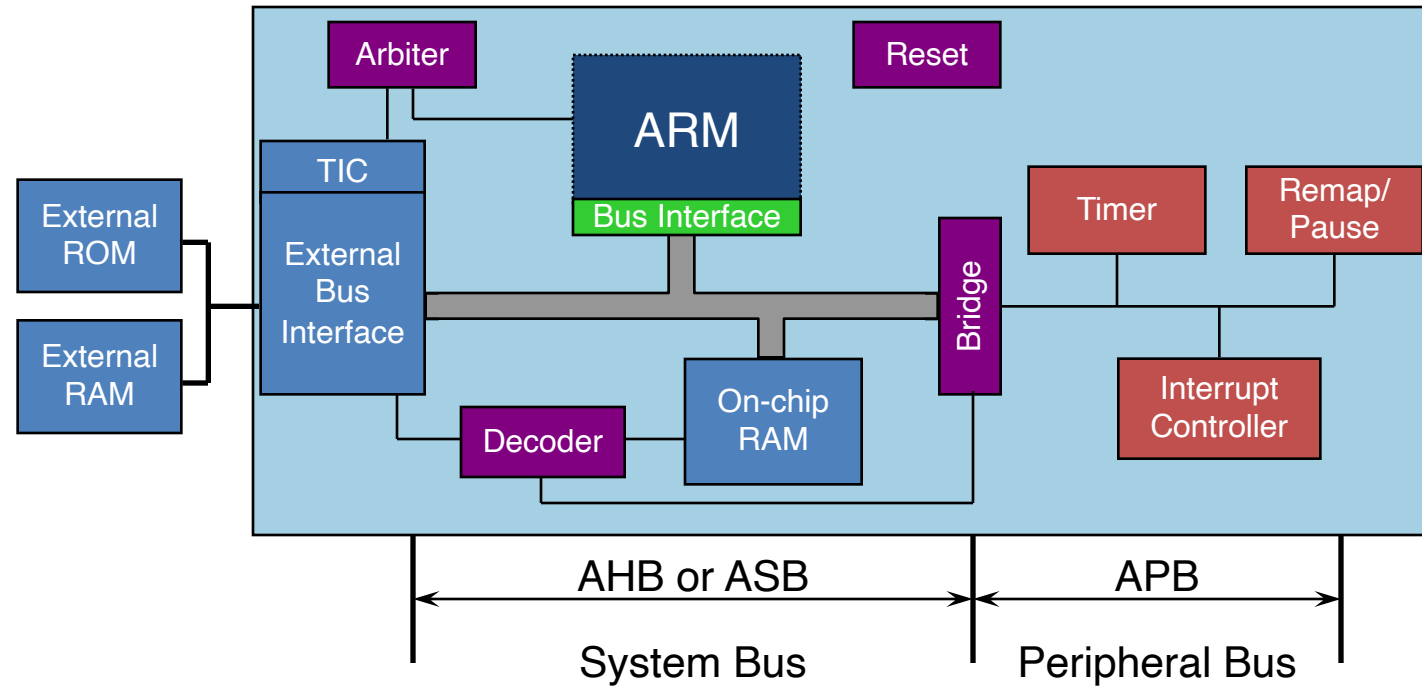
Pentru majoritatea instructiunilor generate de compilator:

- Executia conditionala nu e folosita
- Registrele sursa si destinatie sunt identice
- Doar ultimii 16 biti din registru sunt folositi
- Constantele au marime limitata

# ARM – Exemplu de Arhitectura



# ARM - AMBA



- AMBA (Advanced Microcontroller Bus Architecture)
  1. Standardul ARM de interconectare
  2. Refolosirea IP-core-ului in mai multe proiecte
  3. Permite facilitatea de upgrade si existenta mai multor familii de device-uri

# Unde puteti intalni un ARM?

- Altera Corporation
- Analog Devices Inc
- Atmel
- Broadcom Corporation
- Conexant
- Dolby Labs
- Fraunhofer IIS
- Freescale Semiconductor
- IAR Systems
- Intel Corp
- Mentor Graphics Corporation
- Microsoft
- MSC Vertriebs GmbH
- National Semiconductor
- NEC Engineering
- Nokia
- NVIDIA
- ON Semiconductor
- Samsung Electronics
- Sanyo
- Sun Microsystems Inc
- STMicroelectronics
- Texas Instruments
- Toshiba
- Zilog

# De ce procesoarele ARM sunt cele mai populare de pe piata?

- Low power
  - Solutiile embedded cu ARM ofera cel mai mic consum MIPS/Watt de pe piata la ora actuala.
  - De exemplu, familia STR7 cu ARM7TDMI: 10uA in Stand-by mode
- Cost redus al siliciului
  - Procesoarele ARM si alte produse IP folosesc eficient capabilitatile de procesare si memoria si sunt gandite pentru cerintele de pe piata dispozitivelor wireless.
- Nucleu de calcul performant
  - Arhitecturi de la 1MHz la 1GHz
  - O serie larga de solutii OS, Middleware si tool-uri de programare suportate
  - Foarte multe codecuri multimedia optimizate pentru ARM (ARM Connected Community)