



# Uvod u paralelizam i vektorske instrukcije

(S. Ribarić, Arhitektura i organizacija računarskih sustava, str. 473-532)

- Oblici i razine paralelizma
- Flynnova taksonomija paralelnih arhitektura
- Vektorski procesori i instrukcije
- Vektorske ekstenzije arhitekture x86

Paralelna računala: proširenje Von Neumannovog modela koje omogućava usporedno provođenje različitih koraka obrade.

- sklopolje je pogodno za paralelnu obradu
- međutim, rješenja naših problema najlakše je slijedno formulirati

Oblici raspoloživog paralelizma u računarskim problemima:

- **raspoloživi funkcijski paralelizam** – *različiti* zadatci mogu se izvoditi usporedno (npr. učitavanje i obrada podataka)
- **raspoloživi podatkovni paralelizam** – *isti* zadatci mogu se izvoditi na različitim podatcima  
(npr. tražimo zadani objekt u različitim dijelovima slike)

## Četiri razine (zrnatosti) raspoloživog funkcijskog paralelizma:

- paralelizam na razini instrukcija (ILP) – fino zrnati
  - protočnost, superskalarnost
- paralelizam na razini programskih petlji – srednje zrnati
  - višedretvenost, višejezgrenost (+jezična podrška)
- paralelizam na razini funkcija – srednje zrnati
  - višedretvenost, višejezgrenost (+jezična podrška)
- paralelizam na razini programa – grubo zrnati
  - višejezgrenost

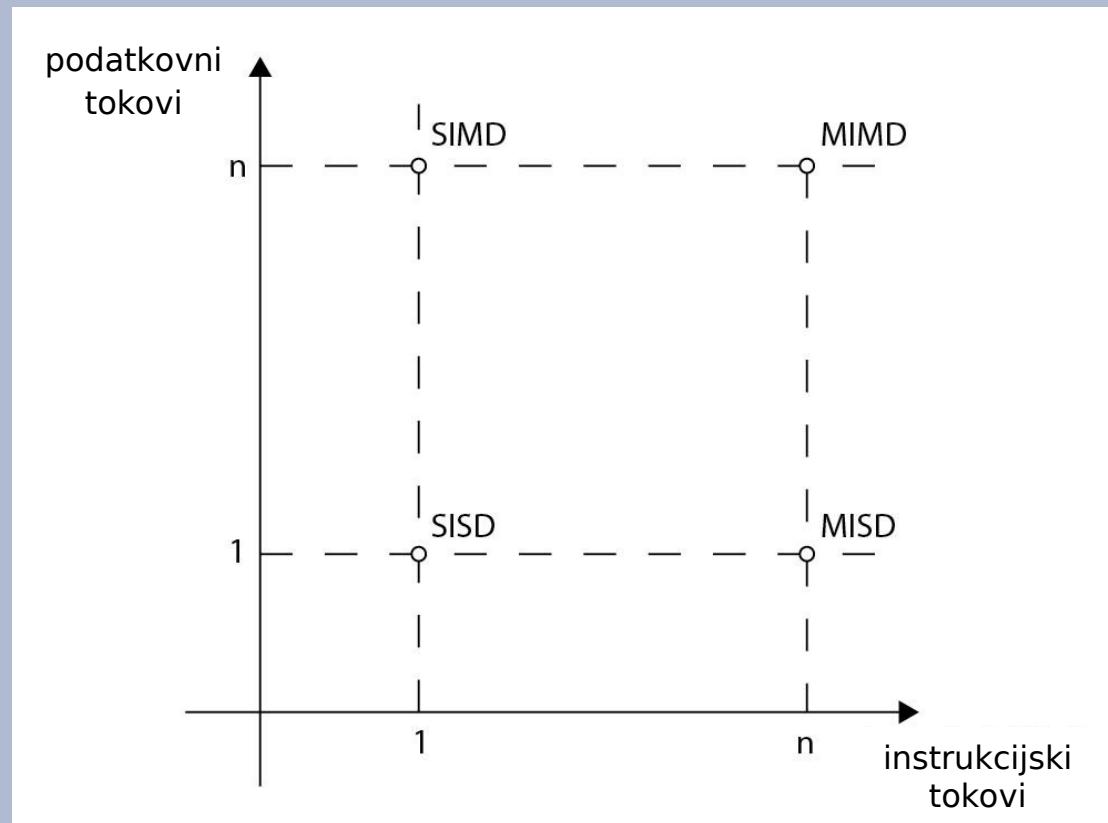
Podatkovni paralelizam može se iskoristiti na dva načina:

- izravno uporabom arhitektura procesora koje podržavaju paralelne operacije na podatkovnim elementima
  - vektorska računala i instrukcije: SIMD, SIMT
  - SIMD: *Single Instruction Stream Multiple Data Stream*
  - SIMT: *Single Instruction, Multiple Threads*
  - SMT: to je nešto sasvim različito!
- pretvorbom raspoloživog podatkovnog paralelizma u funkcijski
  - viši programski jezik označava paralelno izvodljive operacije na podatkovnim elementima (OpenMP)

Računalni sustavi tipa SIMD iskorištavaju podatkovni paralelizam djelovanjem na vektorima s višedimenzionalnim poljima podataka.

Na primjer, instrukcija `mulps zmm1, zmm1, zmm2` (AVX-512) započinje množenje 16 parova operanada i to s latencijom od 4-6 perioda signala vremenskog vođenja.

# Flynnova taksonomija računalnih sustava





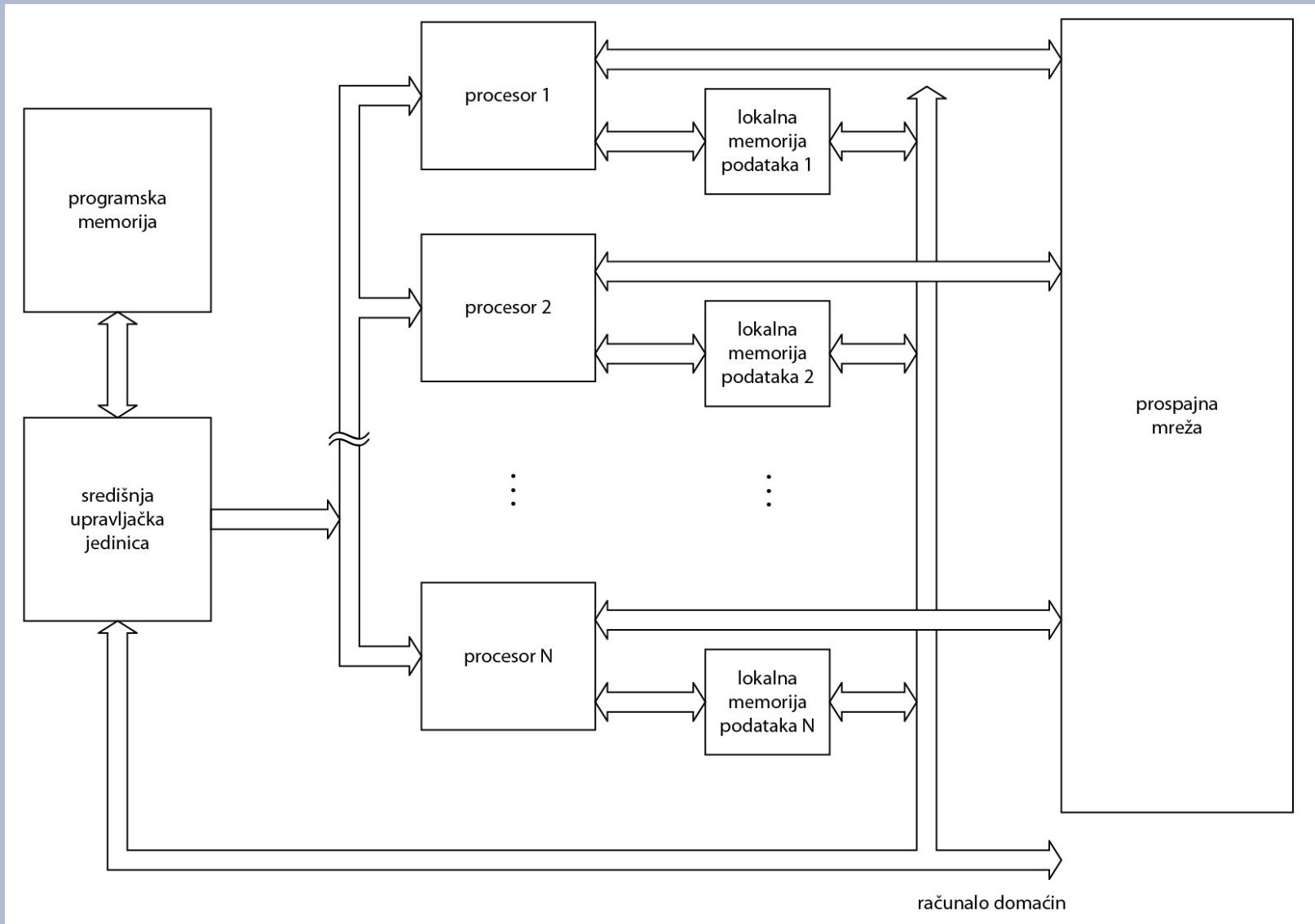
Prema Flynnovoj klasifikaciji, arhitekture paralelnih računalnih sustava su: MISD, SIMD i MIMD

- SIMD - procesori s vektorskim instrukcijama
- MIMD – multiprocesorski sustavi

Osnovna značajka SIMD arhitekture je istodobno izvođenje iste instrukcije na većem broju procesnih jedinica koji djeluju na različitim, višestrukim tokovima podataka.

Takva računala namijenjena su rješavanju problema s visokim stupnjem podatkovnog paralelizma:

- matrično množenje
- obrada jezika, govora, slika, 3D slika, oblaka točaka



Organizacija višeprocesorskog SIMD računarskog sustava

## Vektorski procesori i instrukcije

Jedan od najdjelotvornijih načina iskorištavanja podatkovnog paralelizma postiže se u računalnim sustavima arhitekture SIMD koji se temelje na *vektorskem procesoru* (engl. *vector processor*)

Vektorski procesor obavlja aritmetičke i logičke operacije na operandima koji su vektori.

## Primjer

Razmotrimo računanje sume dvaju 64-dimenzionalnih vektora  $\mathbf{x}$  i  $\mathbf{y}$ .  
Rezultat je vektor  $\mathbf{w}$ :

$$\mathbf{w} = \mathbf{x} + \mathbf{y}.$$

U "običnom" jednoprocesorskom sustavu ta bi se operacija izvela na temelju programskog odsječka:

**for**  $i = 1$  **to** 64

$$w(i) := x(i) + y(i);$$

U vektorskem procesoru gornja bi se operacija izvela samo *jednom vektorskom instrukcijom*, odnosno instrukcijom tipa *vektor-vektor* kojoj su operandi dva 64-dimenzionalna vektora, a rezultat, koji se dobiva u vektorskoj aritmetičko-logičkoj jedinici (vektorska ALU), također je 64-dimenzionalni vektor.

Vektorska ALU može **istodobno zbrojiti/izmnožiti** veći broj odgovarajućih komponenata obaju vektora.

Svaki od vektora, koji predstavlja operand u vektorskoj instrukciji, smješten je u vektorski registar, npr.  $V_i$ , odnosno  $V_j$ , a rezultat se smješta u vektorski registar  $V_k$ .

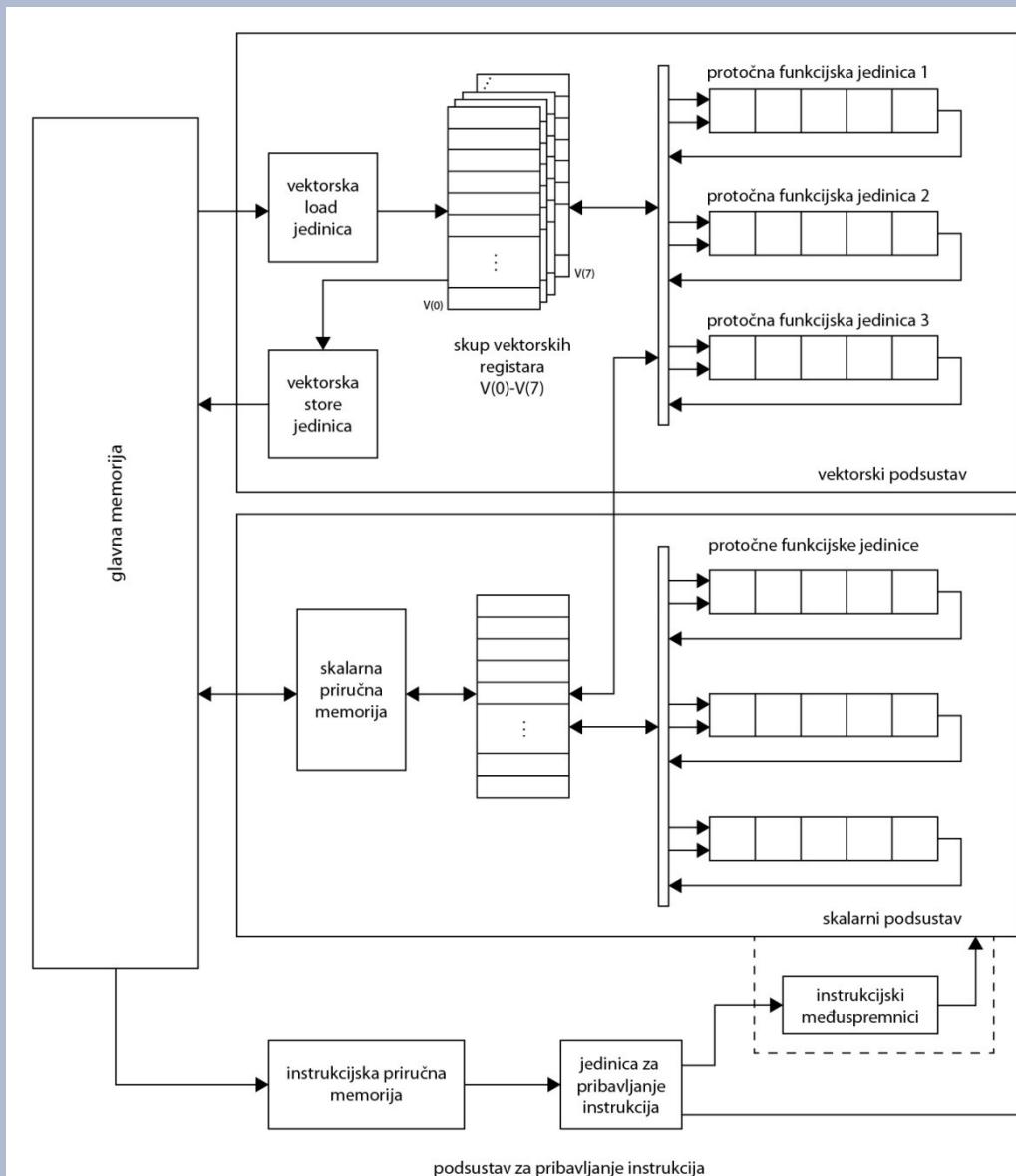
Vektorska instrukcija specificira veliku količinu posla i jednakovrijedna je, u potonjem slučaju, cijeloj programskoj petlji.

Uporaba vektorske ALU u kojoj se istodobno izvode operacije nad svim komponentama vektora u vektorskим je procesorima ipak rijetka.

Umjesto, na primjer, n vektorskih množila  
(koja bi istodobno generirala sve produkte),  
u praksi se koristi protočno množilo  
s relativno velikim brojem protočnih segmenata.

Svaki parcijalni rezultat (umnožak dviju komponenata)  
dobivamo u odgovarajućem koraku obrade  
čije je trajanje određeno trajanjem obrade  
u jednom protočnom segmentu.

Razlozi takva rješenja nisu tehnološka ograničenja,  
već **jednostavnija izvedba**.



Vektorski procesor (pojednostavljeni prikaz)

## Tipovi vektorskih instrukcija

i) instrukcije vektor-vektor:

$$f_1: V_i \rightarrow V_k$$

$$f_2: V_i \times V_j \rightarrow V_k,$$

$V_i$  i  $V_j$  označavaju izvorišni, a  $V_k$  odredišni vektorski registar;

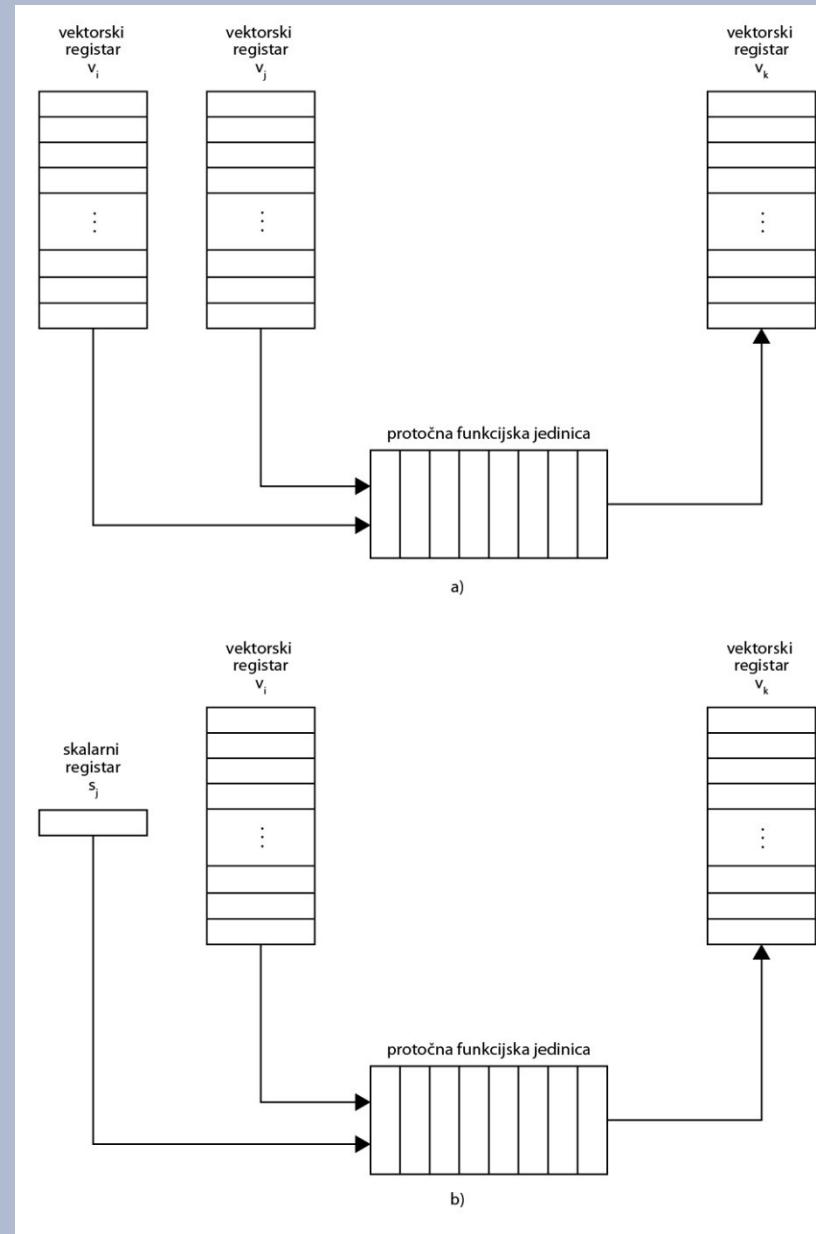
ii) instrukcije vektor-skalar ( $s_j$  označava skalarni registar):

$$f_3: s_j \times V_i \rightarrow V_k,$$

iii) instrukcija vektor-memorija ( $M$  označava radnu memoriju):

$$f_4: M \rightarrow V_j \text{ za operaciju } load$$

$$f_5: V_i \rightarrow M \text{ za operaciju } store,$$



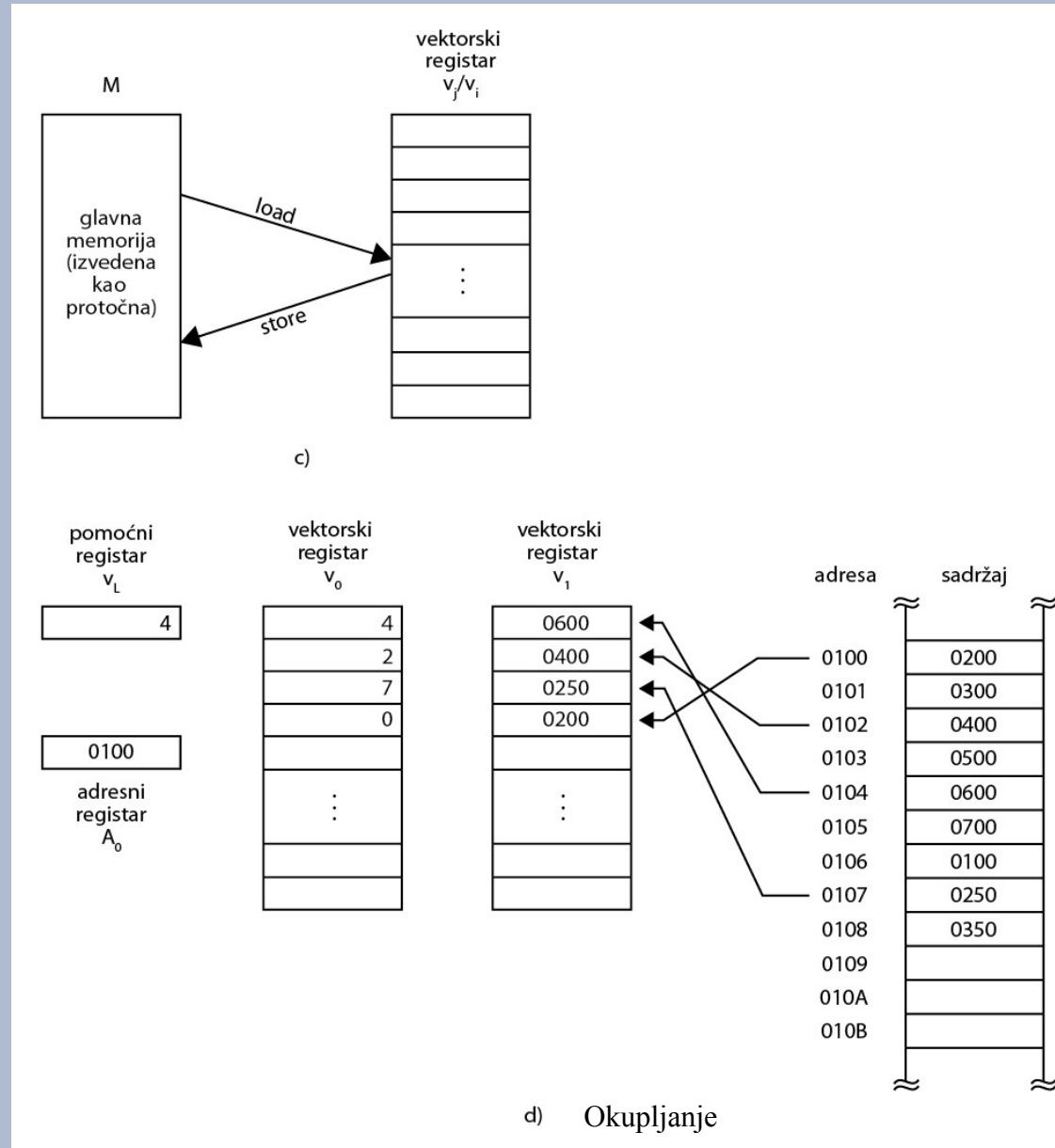
iv) instrukcije redukcije pretvaraju vektore u skalare.

Formalno se taj tip instrukcije može opisati kao:

$$f_6: V_i \rightarrow s_j$$

$$f_7: V_i \times V_j \rightarrow s_j.$$

(npr. dohvat maksimalnog elementa ( $f_6$ ); skalarni produkt ( $f_7$ ))



v) instrukcije okupljanja (engl. *gather*) ili raspršivanja (engl. *scatter*)

$f_8: M \rightarrow V_1 \times V_0$       okupljanje

$f_9: V_1 \times V_0 \rightarrow M$       raspršivanje.

Operacijom okupljanja iz memorije se dohvaćaju elementi različiti od nule, i to tako da vektorski registar  $V_0$  sadržava indekse (kazaljke) na podatke u memoriji, a vektorski registar  $V_1$  sadržava podatke koji se iz memorije dohvaćaju i oblikuju tzv. rijetko popunjeni vektor.

Raspršivanje je obrnuta operacija u odnosu na okupljanje:  
njome možemo u memoriju pohraniti rijetko popunjeni vektor.

vi) instrukcije maskiranja – to su instrukcije vektor-vektor koje dodatno zadaju i vektor maske  $V_m$  (engl. *mask vector*) koji određuje na kojim elementima će se operacija provesti.

Formalno, maskiranje može se opisati kao preslikavanje:

$$f_{10} : V_0 \times V_m \rightarrow V_1.$$

Npr. u vektorskom registru  $V_1$  pohranjuju se elementi регистра  $V_0$  koji su različiti od 0 i za koje je odgovarajući element  $V_m$  različit od nule.

## Primjer

Ilustrirajmo izvođenje operacije  $y = sx + y$

- $y$  i  $x$  su 64 komponentni vektori dvostrukе točnosti (64 bit),
- $s$  je skalar dvostrukе točnosti

Analizu ćemo provesti na skalarnom i vektorskom procesoru.

Pretpostavljamo da su na početku izvođenja  $y$ ,  $x$  i  $s$  pohranjeni u memoriji:

- početna adresa  $x$  je na lokaciji \$s0,
- početna adresa  $y$  je na lokaciji \$s1,
- skalar  $s$  je na lokaciji \$sp.

Programski odsječak za **skalarni procesor** izgleda ovako:

```
ld      $f0, $sp      ; dohvati skalar s i pohrani ga
                  ; u floating-point registar f0
addi   $r4, $s0,#512 ; gornja adresa lokacije na kojoj
                  ; se nalazi vektor x
opet: ld      $f2, 0($s0)    ; dohvati x(i)
      mul   $f2, $f0,$f2    ; s × x(i)
      ld      $f4, 0($s1)    ; dohvati y(i)
      add   $f4, $f2, $f4    ; s × x(i) + y(i)
      st      $f4, 0($s1)    ; pohrani rezultat na y(i)
      addi   $s0, $s0, #8     ; uvećaj indeks za x
      addi   $s1, $s1, #8     ; uvećaj indeks za y
      sub    $t0, $r4, $s0     ; razlika tekuće i krajnje adrese
      bne   $t0, $zero, opet ; petljaj dok razlika ne postane =0
```



(Opaska: adresna zrnatost memorije je bajtna, zato je gornja granica 512, tj.  $64 \times 8$ , gdje je 64 broj komponenti vektora, a svaka je njegova komponenta duljine 8 bajtova.)

Programski odsječak za **vektorski procesor** izgleda ovako:

|       |                 |  |
|-------|-----------------|--|
| ld    | \$f0, \$sp      | ; dohvati skalar <i>s</i> i pohrani ga u<br>; floating-point registar f0 |
| ldv   | \$v1, 0(\$s0)   | ; dohvati vektor <b>x</b> i pohrani ga u<br>; vektorski registar v1      |
| mulvs | \$v2, \$v1,\$f0 | ; množenje vektora <b>x</b> sa<br>; skalarom <i>s</i>                    |
| ldv   | \$v3, 0(\$s1)   | ; dohvati vektor <b>y</b> i pohrani ga u<br>; vektorski registar v3      |
| addv  | \$v4, \$v2,\$v3 | ; pribroji <b>y</b> produktu <i>sx</i>                                   |
| stv   | \$v4, 0(\$s1)   | ; pohrani rezultat   |

## Usporedba gornjih programskih odsječaka:

- vektorski program **značajno reducira promet instrukcija** – on zahtijeva samo šest instrukcija, dok se kod skalarнog procesora zahtijeva skoro 600 instrukcija (programska petlja).
- druga zanimljiva razlika je u učestalosti hazarda – za skalarni procesor svaka *add* instrukcija mora čekati na *mul* instrukciju te svaka *st* instrukcija mora čekati na *add* instrukciju.
- za vektorski će procesor svaka vektorska instrukcija biti u zastaju samo za prvi element svakog od vektora, tako da će ostali elementi glatko protjecati kroz protočnu strukturu.

Vektorske instrukcije prvo su korištene u tzv.  
superračunalima (npr. Cray 1, 1976)

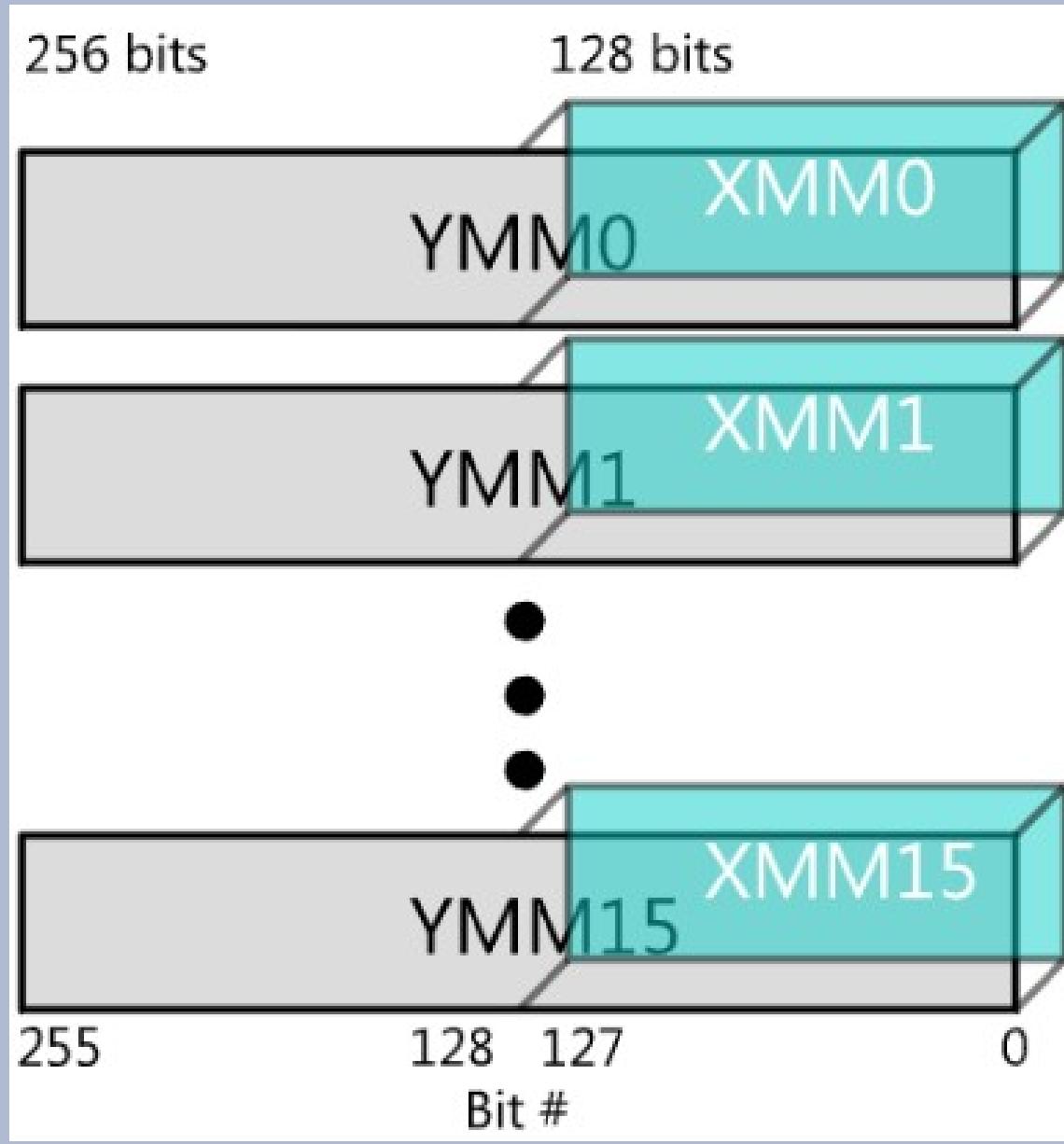
U novije vrijeme, koriste se i u računalima opće namjene  
(x86: MMX, SSE, AVX, ARM: Neon)

U nastavku ćemo ukratko predstaviti vektorska proširenja za  
arhitekturu **x86** (MMX, SSE, AVX)

Primjeri će koristiti Intelovu sintaksu (na gcc-u treba  
zadati `.intel_syntax noprefix`)

## Vektorski registri ekstenzija arhitekture x86

- xmm (SSE):  $16 \times 128 \text{ bit} = 16 \times 4 \times \text{fp32}$ ,
- ymm (AVX):  $16 \times 256 \text{ bit} = 16 \times 8 \times \text{fp32}$ ,
- zmm (AVX-512):  $32 \times 512 \text{ bit} = 32 \times 16 \times \text{fp32}$
- donja polovica od ymm je xmm; jednako za zmm i ymm.



## Vektorske instrukcije (SSE):

| Data transfer                      | Arithmetic                   | Compare          |
|------------------------------------|------------------------------|------------------|
| MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm | ADD{SS/PS/SD/PD} xmm,mem/xmm | CMP{SS/PS/SD/PD} |
|                                    | SUB{SS/PS/SD/PD} xmm,mem/xmm |                  |
| MOV {H/L} {PS/PD} xmm, mem/xmm     | MUL{SS/PS/SD/PD} xmm,mem/xmm |                  |
|                                    | DIV{SS/PS/SD/PD} xmm,mem/xmm |                  |
|                                    | SQRT{SS/PS/SD/PD} mem/xmm    |                  |
|                                    | MAX {SS/PS/SD/PD} mem/xmm    |                  |
|                                    | MIN{SS/PS/SD/PD} mem/xmm     |                  |

[patterson13]

A/U: aligned/**unaligned**

S/P: scalar/**packed**

S/D: **single precision**/double precision

H/L: high half/low half

## Primjeri

Zbroji vektore na koje pokazuju `rsi` i `rdx` te upiši rezultat na lokaciju na koju pokazuje `rdi`:

```
movups xmm1, WORD PTR [rsi]
movups xmm2, WORD PTR [rdx]
addps xmm1, xmm1, xmm2
movups WORD PTR [rdi], xmm1
```

Pomnoži `ymm1` i `ymm3` po elementima, dodaj `ymm2` i spremi natrag u `ymm3` (fused multiply-add):

```
fmadd ymm1, ymm2, ymm3
```

Postavi memorijski operand u sve elemente vektorskog registra (vector broadcast):

```
vbroadcastss ymm1, WORD PTR [rax+r8]
```

# Arhitektura računala 2

## Performansa računala

Siniša Šegvić i Slobodan Ribarić

Fakultet elektrotehnike i računarstva

Sveučilište u Zagrebu

## UVOD: PODSJETIMO SE ŠTO SMO RADILI DO SADA

Klasična arhitektura računala:

- Von Neumannovo računalo
- Pojednostavljeni model procesora
- Ožičena izvedba upravljačke jedinice
- Mikroprogramirana izvedba upravljačke jedinice
- Sučelje prema programskoj podršci

# UVOD: ŠTO ĆEMO RAZMATRATI SADA?

Arhitektura i organizacija modernih računala

- Performansa računala
- Priručne memorije
- Put podataka protočne arhitekture MIPS
- Procesori s višestrukim izdavanjem
- Virtualna memorija
- Višeprocesorski sustavi

## PERFORMANSA: VRIJEME ODZIVA I PROPUSTNOST

Prisjetimo se, arhitektura računala razmatra **performansu**, cijenu, utrošak energije, pouzdanost, raspoloživost

Dvojako značenje performanse računala:

1. **vrijeme odziva** (response time, za sada razmatramo to!):
  - koliko vremena je potrebno za obavljanje zadatka?
  - uglavnom relevantno kod radnih stanica i ugrađenih primjena
2. **propusnost** (throughput):
  - ukupni obavljeni posao u jedinici vremena (npr, transakcija/min)
  - uglavnom relevantno kod poslužitelja

Kako na performansu utječe zamjena procesora bržim modelom?

A nabava dodatnog procesora?

## PERFORMANSA: DEFINICIJA I RELATIVNA PERFORMANSA

Performansa  $P \triangleq (\text{vrijeme izvođenja})^{-1}$

Što znači da računalo A ima  $n$  puta veću performansu od računala B za zadani program  $X$ ?

$$\frac{P_A}{P_B} = n \Rightarrow \frac{t_B}{t_A} = n$$

Najčešće će nas interesirati upravo relativna performansa  $p$ :

$$p_A(B) \triangleq \frac{P_A}{P_B} = \frac{t_B}{t_A}$$

Npr, izmjerimo trajanje izvođenja programa X na računalima A i B:

- $t_A(X) = 10 \text{ s}, t_B(X) = 15 \text{ s}$
- $\Rightarrow p_A(B, X) = \frac{t_B(X)}{t_A(X)} = 1,5$
- Računalo A je za program X 1,5 puta brže!

## PERFORMANSA: AMDAHLOV ZAKON

Amdahlov zakon opisuje poboljšanje performanse uslijed ubrzavanja samo jednog segmenta obrade. Neka je zadano:

- $s$  faktor ubrzanja promatranog segmenta obrade
- $x$  udio razmatranog dijela u cijelom postupku

Koliko bi tada bilo ubrzanje  $p$ ?

$$p = \frac{T_{\text{staro}}}{(1 - x) \cdot T_{\text{staro}} + x/s \cdot T_{\text{staro}}} = \frac{1}{(1 - x) + x/s}$$

**Primjer:** pretpostavimo da se u nekom algoritmu na množenje troši 80% vremena.

Koliko trebamo poboljšati množenje ako želimo algoritam ubrzati 5 puta?

**Odgovor:** zadatak nije moguće riješiti!

## PERFORMANSA: AMDAHLOV PRIMJER

Razmatramo nabavu novog procesora za 8 godina stari producijski poslužitelj:

- poznato je da poslužitelj 60% vremena čeka na diskove i mrežu
- performansa novog procesora je  $10\times$  veća od starog
- koliki bi bio učinak akvizicije?

$$p = \frac{1}{(1-x) + x/s} = \frac{1}{0.6 + 0.4/10} \doteq 1.56$$

**Odgovor:** 56%.

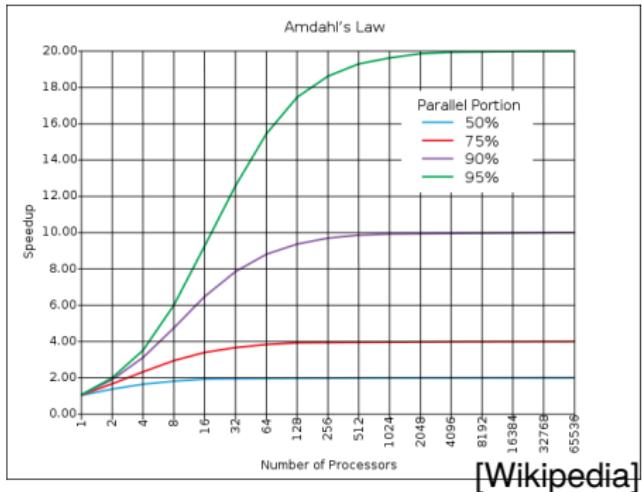
Pokazuje se da kod ovakvih proračuna ljudi često griješe. Amdahlov zakon nam pomaže da ostvarimo bolji kontakt sa stvarnošću.

# PERFORMANSA: AMDAHL ZA PARALELNE IZVEDBE

Koliko možemo ubrzati algoritam na paralelnom računalu ako slijedni udio iznosi  $w$  ( $w = 1 - x$ )?

Amdahlov zakon: maksimalno  $1/w$ .

Desno: graf ubrzanja u ovisnosti o broju procesora  $s$



[Wikipedia]

Ograničenja ovog modela:

- ukupan posao u paralelnom algoritmu je tipično veći nego u ekvivalentnom slijednom postupku
- ipak, performansa paralelnog sustava može rasti i superlinearno zbog veće količine akumulirane priručne memorije

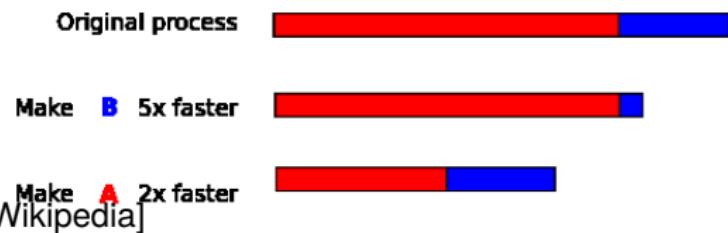
## PERFORMANSA: POSLJEDICE AMDAHLA

Amdahlov zakon kaže da ne možemo očekivati ukupno poboljšanje proporcionalno poboljšanju samo jednog aspekta obrade.

Glavna posljedica Amdahlovog zakona:

resurse valja alocirati za poboljšanje najčešćeg slučaja  
(engl. make common case fast)

Two Independent parts    **A**   **B**



# PERFORMANSA: O MJERENJU TRAJANJA IZVOĐENJA

Više načina za mjerjenje vremena izvođenja programa

- **ukupno vrijeme izvođenja** uključuje sve aspekte obrade:
  - najizravnija mjera performanse, determinira performansu sustava
  - čimbenici: CPU, RAM, U-I (mreža, disk), OS, ...
- **trajanje izvođenja procesa** određeno procesorom i memorijom
  - zanemaren utjecaj U-I i ostalih procesa
  - vrijeme procesa obuhvaća:
    - ◊ vrijeme korisničkog programa  $t_{CPU}$
    - ◊ vrijeme operacijskog sustava  $t_{OS}$
    - ◊ za detalje pogledati `man time` na UNIX-ima
  - različiti programi imaju različit omjer  $t_{CPU}/t_{OS}$ , ali najčešće je ipak  $t_{CPU} \gg t_{OS}$
- U nastavku ćemo detaljnije promotriti  $t_{CPU}$

## PERFORMANSA: RADNA FREKVENCIJA PROCESORA

Uloga frekvencije signala takta procesora  $f_{\text{CPU}}$ :

- digitalni sklopovi izvode mikrooperacije u diskretnim periodima signala takta
- trajanje takta određeno najsporijom mikrooperacijom
- npr,  $t_{\mu_{op}} = 250\text{ps} \Rightarrow f_{\text{CPU}} = 4\text{ GHz}$
- u načelu, brži takt implicira veću performansu, ali svi znamo da veza nije jednostavna
  - računala rade na 2 – 3 GHz već 5 godina, ali performansa ipak raste oko 20% godišnje ( $1,2^5 \approx 2,5!$ )
  - ⇒ danas računala naprave više u jednom periodu (ciklusu) signala takta nego prije 5 godina

## PERFORMANSA: UTJECAJ RADNE FREKVENCIJE

Formalizirajmo vezu između procesorskog vremena i radnog takta:

$$t_{\text{CPU}} = n_{\text{ciklusa}} \cdot T_{\text{takt}} = \frac{n_{\text{ciklusa}}}{f_{\text{CPU}}}$$

Vidimo da će performansi pogodovati:

- smanjenje broja taktova pri izvođenju programa
- povećanje radne frekvencije

Arhitekti obično traže optimalan kompromis između radne frekvencije  $f_i$  i broja ciklusa  $n_c$  ...

... osim ako se u igru ne umiješa marketinški odjel :-)

## PERFORMANSA: $t = n_i/f$

Pretpostavimo da na tržištu postoji računalo A koje uz  $f_A = 2 \text{ GHz}$  ispitni program izvodi  $t_A = 10 \text{ s}$ .

Pretpostavimo da nam je ambicija projektirati računalo B uz  $t_B = 6 \text{ s}$ .

Znamo da možemo postići znatno brži takt, ali po cijenu da  $n_{cB}$  bude jednak  $1,2 \cdot n_{cA}$ .

Kolika mora biti radna frekvencija računala B?

Rješenje:

- $n_{cA} = t_A \cdot f_A$
- $f_B = \frac{n_{cB}}{t_B} = 1,2 \cdot \frac{t_A}{t_B} \cdot f_A = 4 \text{ GHz}$

## PERFORMANSA: NC = NI × CPI

U jednadžbu performanse uvodimo broj instrukcija  
(možemo mjeriti, za postojeće programe je konstantan).

CPI --- broj ciklusa po instrukciji, engl. cycles per instruction  
(veza između broja taktova  $n_c$  i broja instrukcija  $n_i$ ):

$$n_c = n_i \cdot \text{CPI}$$

Konačni oblik jednadžbe procesorske performanse:

$$t_{\text{CPU}} = \frac{n_i \cdot \text{CPI}}{f_{\text{CPU}}} = n_i \cdot \text{CPI} \cdot T_{\text{takt}}$$

Još jedan zapis jednadžbe:

$$t_{\text{CPU}} = \text{instrukcije} \cdot \frac{\text{ciklusi}}{\text{instrukcija}} \cdot \frac{\text{sekunde}}{\text{ciklus}}$$

## PERFORMANSA: DETALJI

- $n$ ; ... ukupan broj instrukcija koje se izvedu u okviru izvođenja programa, ovisi o:
  - **problemu**
  - inventivnosti prevoditelja
  - instrukcijskoj arhitekturi procesora
- CPI ... prosječan broj ciklusa po instrukciji
  - ovisi o arhitekturi i **organizaciji** procesora
  - ako CPI nije konstantan (najčešće nije) uzimamo ponderiranu srednju vrijednost (težinske faktore određujemo empirijski)
- $f$  ... frekvencija radnog takta, ovisi o:
  - **tehnologiji** izvedbe integriranog sklopa
  - organizaciji procesora

## PERFORMANSA: PRIMJER

Procesor A: period takta: 250 ps, CPI=2,0

Procesor B: period takta: 500 ps, CPI=1,2

Procesori imaju istu instrukcijsku arhitekturu ( $n_{iA} = n_{iB} = n_i$ )

Zadatak: usporediti performansu dvaju procesora.

Rješenje:

- $t_A = n_i \cdot \text{CPI}_A \cdot T_A$
- $t_B = n_i \cdot \text{CPI}_B \cdot T_B$
- $p_A(B) = \frac{t_B}{t_A} = \frac{600}{500} = 1,2$

Procesor A na 4 GHz je 20% jači od procesora B na 2 GHz

## PERFORMANSA: CPI, DETALJNIJE

Kod modernih računala CPI ovisi o tipu instrukcije  
(latencije ADD, MULSS, DIVSS: 1, 4, 14)

Kod modernih računala CPI ne ovisi samo o tipu instrukcije, nego i o dinamičkim uvjetima u trenutku izvođenja

- ipak, za svaki razred instrukcija  $r$  može se procijeniti očekivani CPI $_r$
- tipovi: zbrajanje/oduzimanje, memorijske, grananje, ...

Nadalje, za relevantni skup programa, empirijskim metodama možemo utvrditi diskretnu distribuciju razreda instrukcija  $p_r$  (vrijedi  $\sum_r p_r = 1$ )

Tada CPI procesora možemo procijeniti kao:

$$\text{CPI} = \sum_r p_r \cdot \text{CPI}_r$$

## PERFORMANSA: CPI, PRIMJER

Zadana su dva alternativna prijevoda (1, 2) istog programa. Prijevodi koriste instrukcije iz razreda A, B i C. Ocijeniti CPI za oba prijevoda.

| razred $r$ | A | B | C |
|------------|---|---|---|
| CPI $_r$   | 1 | 2 | 3 |
| $n_{i1}$   | 2 | 1 | 2 |
| $n_{i2}$   | 4 | 1 | 1 |

$$\text{CPI}_1 = \frac{2}{5} \cdot 1 + \frac{1}{5} \cdot 2 + \frac{2}{5} \cdot 3 = 2 \quad (n_{c1} = 10)$$

$$\text{CPI}_2 = \frac{4}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 = 1,5 \quad (n_{c2} = 9)$$

## PERFORMANSA: CPI, PRIMJER 2

Pretpostavimo da smo napravili sljedeća mjerena:

- učestalost i prosječni CPI operacija s pomičnim zarezom: 25%, 4
- prosječan CPI svih ostalih operacija: 1.33
- učestalost, CPI instrukcije FPSQR: 2%, 20

Usporediti sljedeće razvojne alternative (pretp. isti utjecaj na  $f$ ):

1. smanjiti CPI instrukcije FPSQR na 2
2. smanjiti CPI **svih** FP instrukcija na 2,5

**Rješenje:** 1. primijetiti:  $n_i$  ostaje isti (novi CPU, stari programi!)

$$2. \text{ CPI}_{orig} = 0,75 \cdot 1,33 + 0,25 \cdot 4 = 2$$

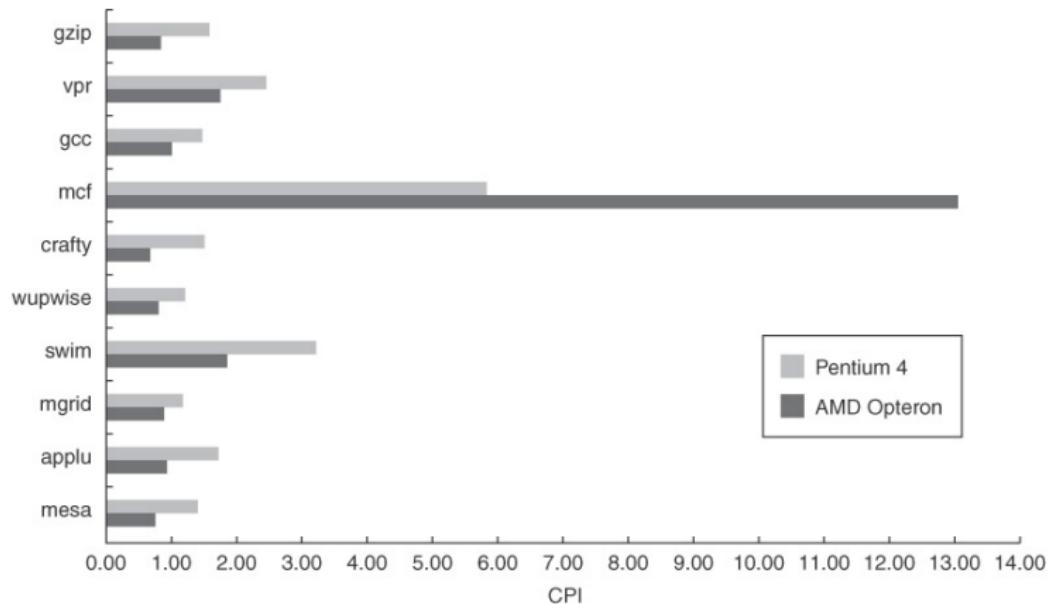
$$3. \text{ CPI}_1 = \text{CPI}_{orig} - 2\% \cdot 20 + 2\% \cdot 2 = 1,64$$

$$4. \text{ CPI}_2 = \text{CPI}_{orig} - 25\% \cdot 4 + 25\% \cdot 2,5 = 1,63$$

# PERFORMANSA: CPI U PRAKSI

CPI možemo **mjeriti** tijekom izvođenja reprezentativnih programa:

$$\text{CPI} = \frac{t_{\text{CPU}} \cdot f_{\text{CPU}}}{n_i}$$



## PERFORMANSA: MEĐUSAŽETAK

- A je n puta brži od B ako:

$$n = P_A(B) = \frac{P_A}{P_B} = \frac{t_B}{t_A}$$

- U procesorski intenzivnim aplikacijama (bez U-I, VM, OS) vrijedi:

$$t_{\text{CPU}} = n_i \cdot \text{CPI} \cdot T_{\text{takt}} = \text{broj instrukcija} \cdot \frac{\text{ciklusi}}{\text{instrukcija}} \cdot \frac{\text{sekunde}}{\text{ciklus}}$$

- Performansa ovisi o:
  - problemu (glavni utjecaj na  $n_i$ )
  - prevoditelju ( $n_i$ , CPI)
  - instrukcijskoj arhitekturi ( $n_i$ , CPI,  $f$ )
  - sklopovskoj organizaciji (CPI,  $f$ )
  - tehnologiji ( $f$ )

## PERFORMANSA: EVALUACIJA

Kako evaluirati  $t_{\text{CPU}}^{-1}$ ,  $t_{\text{CPU}}^{-1} W^{-1}$ , mrežu, baze, ...?

- trebaju nam referentni evaluacijski programi (engl. benchmark)
- evaluacijski programi mogu biti:
  - važni stvarni programi (gcc, perl, gzip),
  - sintetički programi koji simuliraju stvarna opterećenja (npr, dhystone, whetstone, CoreMark...)
- kako bi se povećala stabilnost, obično se koriste evaluacijske kolekcije (engl. benchmark suites)
- Ozbiljnije evaluacijske kolekcije održavaju specijalizirane tvrtke:
  - Standard Performance Evaluation Corporation (SPEC)
  - Transaction Processing Performance Council (TPC)
  - Embedded Microprocessor Benchmark Consortium (EEMBC)
  - ...

## PERFORMANSA: SPEC

Standard Performance Evaluation Corporation (<http://www.spec.org>):

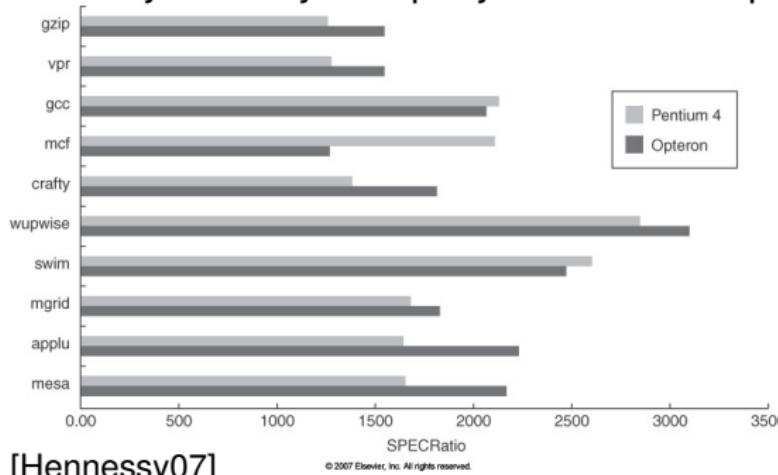
- održava i razvija evaluacijske programe za računala opće namjene
- procesori, mrežni servisi, performansa/snaga, Java, itd.
- ocjene (SPECmark) odgovaraju **relativnoj performansi**
- lako se pokaže da odabir **referentnog računala** nije važan

Procesorske kolekcije ciljaju cijelobrojnu (CINT) i FP performansu (CFP)

- mjeri se **brzina** (CINT2006) i **propusnost** (CINT2006\_rate)
  - propusnost odgovara trajanju usporednih instanci istog programa
- za svaki test dodjeljuju se dvije ocjene:
  - **osnovna** (base) ⇒ iste opcije prevoditelja za cijelu kolekciju
  - **vršna** (peak) ⇒ opcije prilagođene pojedinačnim testovima

## PERFORMANSA: ZBIRNA OCJENA

Pokazuje se da je rasipanje rezultata na pojedinim testovima veliko:



[Hennessy07]

© 2007 Elsevier, Inc. All rights reserved.

⇒ ukupnu ocjenu određujemo kao **geometrijsku sredinu** pojedinačnih (aritmetička sredina omjera nema smisla):

$$P_{X,\text{SPEC}}(R) = \sqrt[n]{\prod_i P_{X,i}(R)}$$

## PERFORMANSA: SPEC 2006, PRIMJER

Koji je procesor bolji odabir za specifične (*peak*) aplikacije?

| CPU (SPECmark peak)     | cijena  | CINT | CFP | CINTr | CFPr |
|-------------------------|---------|------|-----|-------|------|
| Intel Core 2 Quad Q8400 | 1300 kn | 22   | 20  | 59    | 41   |
| AMD Phenom II X4 955    | 1400 kn | 19   | 19  | 53    | 43   |

U ovom trenutku Intel nudi više performanse za novac.

# PERFORMANSA: SPEC-ovi PROGRAMI

| SPEC2006 benchmark description           | SPEC2006   | Benchmark name by SPEC generation |        |        |           |
|--|------------|-----------------------------------|--------|--------|-----------|
|  |            | SPEC2000                          | SPEC95 | SPEC92 | SPEC89    |
| GNU C compiler                           |            |                                   |        |        | gcc       |
| Interpreted string processing            |            | perl                              |        |        | espresso  |
| Combinatorial optimization               | mcf        |                                   |        |        | li        |
| Block-sorting compression                | bzip2      |                                   |        |        | eqntott   |
| Go game (AI)                             | go         | vortex                            |        |        |           |
| Video compression                        | h264avc    | gzip                              |        |        |           |
| Games/path finding                       | astar      | eon                               |        |        |           |
| Search gene sequence                     | hmmer      | twolf                             |        |        |           |
| Quantum computer simulation              | libquantum | vortex                            |        |        |           |
| Discrete event simulation library        | omnetpp    | vpr                               |        |        |           |
| Chess game (AI)                          | sjeng      | crafty                            |        |        |           |
| XML parsing                              | xalancbmk  | parser                            |        |        |           |
| CFD/blast waves                          | bwaves     |                                   |        |        | fpppp     |
| Numerical relativity                     | cactusADM  |                                   |        |        | tomcatv   |
| Finite element code                      | calculix   |                                   |        |        | doduc     |
| Differential equation solver framework   | dealII     |                                   |        |        | nasa7     |
| Quantum chemistry                        | gamess     |                                   |        |        | spice     |
| EM solver (freq/time domain)             | GemsFDTD   |                                   |        |        | matrix300 |
| Scalable molecular dynamics (~NAMD)      | gromacs    |                                   |        |        |           |
| Lattice Boltzman method (fluid/air flow) | lmb        |                                   |        |        |           |
| Large eddie simulation/turbulent CFD     | LESlie3d   | wupwise                           |        |        |           |
| Lattice quantum chromodynamics           | milc       | apply                             |        |        |           |
| Molecular dynamics                       | namd       | galgel                            |        |        |           |
| Image ray tracing                        | povray     | mesa                              |        |        |           |
| Spare linear algebra                     | soplex     | art                               |        |        |           |
| Speech recognition                       | sphinx3    | equake                            |        |        |           |
| Quantum chemistry/object oriented        | tono       | facerec                           |        |        |           |
| Weather research and forecasting         | wrf        | ammp                              |        |        |           |
| Magneto hydrodynamics (astrophysics)     | zeusmp     | lucas                             |        |        |           |
|  |            | fma3d                             |        |        |           |
|  |            | sixtrack                          |        |        |           |

[Hennessy07]

## PERFORMANSA: SPEC-OVI PROGRAMI

Prethodna slika prikazuje evoluciju kolekcija SPEC CINT i SPEC CFP

Referentni programi se moraju odabratи tako da kolekcija odražava  
**tipična** stvarna opterećenja

Dodatno, proizvođači se **prilagođavaju** referentnim programima, pa je  
programe često potrebno mijenjati i prije nego što zastare

Razdioba programskih jezika u CINT 2006: (C: 9; C++: 3)

Razdioba programskih jezika u CFP 2006: (Fortran: 6; C++: 4; C: 3;  
C/Fortran: 4)

## PERFORMANSA: POUKE

Jednadžba procesorske performanse:

$$t_{\text{CPU}} = n_i \cdot \text{CPI} \cdot T_{\text{takt}}$$

Kako povećati performansu u okvirima postojeće tehnologije?

- usredotočiti se na najčešći slučaj ( $\text{CPI} \cdot T_{\text{takt}} \downarrow$ ):  
npr, prilagoditi put podataka najvažnijim mikrooperacijama,
- iskoristiti paralelizam ( $\text{CPI} \downarrow$ ):  
npr, predvidjeti dobro popunjenu protočnu strukturu
- iskoristiti lokalnost pristupa memoriji ( $T_{\text{takt}} \downarrow$ ):  
npr, predvidjeti registarske skupove i priručne memorije

Ključna tri načela pri projektiranju sklopova s kompetitivnom performansom!

## PERFORMANSA: NAJČEŠĆI SLUČAJ

**Načelo:** kad se pojavi potreba za kompromisom, favoriziraj česti slučaj

- npr, pribavljanje instrukcije češće od množenja  
⇒ optimirati fazu pribavljanja!
- npr, množenje češće od dijeljenja (pristup elementu matrice)  
⇒ množenju alocirati više tranzistora!

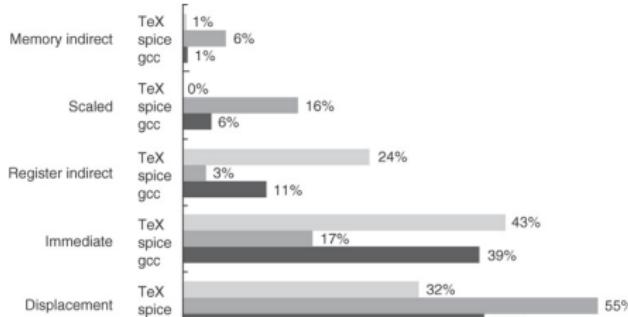
Kako se favoriziranje čestog slučaja preslikava u ukupnu performansu?

→ Amdahlov zakon!

# PERFORMANSA: NAJČEŠĆI I NAJJEDNOSTAVNIJI

Česti slučaj **obično** ujedno i jednostavniji pa se može implementirati brže

- pokazuje se da se složeni načini adresiranja rijetko koriste  
⇒ treba optimirati jednostavne načine adresiranja
- to će nekoliko usporiti slučajeve u kojima se složeni načini adresiranja *mogu* koristiti, ali ukupna performansa će se povećati!
- usputno, registarsko indirektno, te registarsko indirektno adresiranje s pomakom su na VAX-u tipično zastupljeni s preko 75% (slika!)



## PERFORMANSA: PARALELIZAM

**Načelo:** povećati performansu usporednom obradom

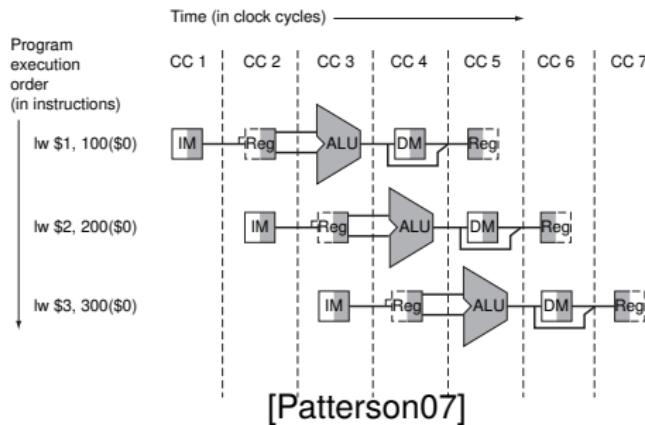
- npr, povećati propusnost poslužitelja povećanjem broja procesora ili diskova
- npr, povećati memorijsku propusnost prepletenim spremanjem podataka na više sklopova
- protočno i superskalarno izvršavanje instrukcija

Protočnost je posebno efikasan način povećanja performanse

- etape susjednih instrukcija najčešće se mogu izvršavati usporedno (ponovo se fokusiramo na najčešći slučaj)
- CPI  $\downarrow \times 4$

# PERFORMANSA: PARALELIZAM, PRIMJER

Protočno izvođenje instrukcija u arhitekturi MIPS:



- postiže se  $CPI \approx 1$ , iako latencija svake instrukcije iznosi 5 ciklusa,
- pokazuje se da je ideja ograničena međuvisnostima među instrukcijama (hazardima)
- iako postoje, dublje protočne strukture najčešće nisu opravdane (npr, Pentium 4 Prescott - 30 segmenata)

## PERFORMANSA: LOKALNOST

**Načelo:** iskoristiti lokalnost pristupa memoriji

- još jedan važan empirijski potvrđen česti slučaj
- veliki registarski skupovi i priručne memorije:  
→ lijek za Von Neumannovo usko grlo!
- postoji prostorna i vremenska lokalnost

Usporedimo vremena potrebna za izvođenje tipičnih operacija:

- istovremeno čitanje dva registra te upisivanje u treći (1 T)
- pristup priručnoj memoriji L1 (1 T), L2 (5 T) i L3 (25 T)
- pristup glavnoj memoriji (100 T)
- pristup disku ( $10^6$  T)

## PERFORMANSA: SAŽETAK

Jednadžba procesorske performanse:

$$t_{\text{CPU}} = n_i \cdot \text{CPI} \cdot T_{\text{takt}}$$

Procesorsku performansu mjerimo kolekcijama pažljivo odabralih ispitnih programa

Moderna načela za ostvarivanje visoke performanse:

- usredotočiti se na najčešći slučaj
- iskoristiti paralelizam
- iskoristiti načelo lokalnosti

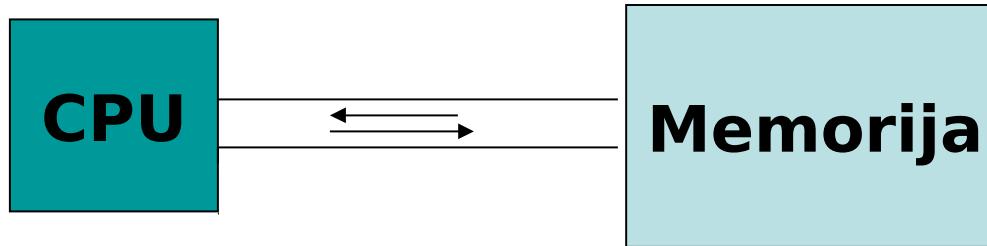
## PERFORMANSA: LITERATURA

1. Computer Organization and Design, 4th ed, David Patterson and John Hennessy, Morgan Kaufmann
2. Computer Architecture: A Quantitative Approach, 4th ed, John Hennessy and David Patterson Morgan Kaufmann
3. Arhitektura računala CISC i RISC, Slobodan Ribarić, Školska knjiga 1996

# **6. Priručne memorije**

1. Svojstva i organizacija dinamičkog RAM-a
2. Memorijska hijerarhija
3. Organizacija priručne memorije
4. Odabir parametara, performansa
5. Izvedbeni detalji

# Komunikacija s memorijom usko grlo performanse



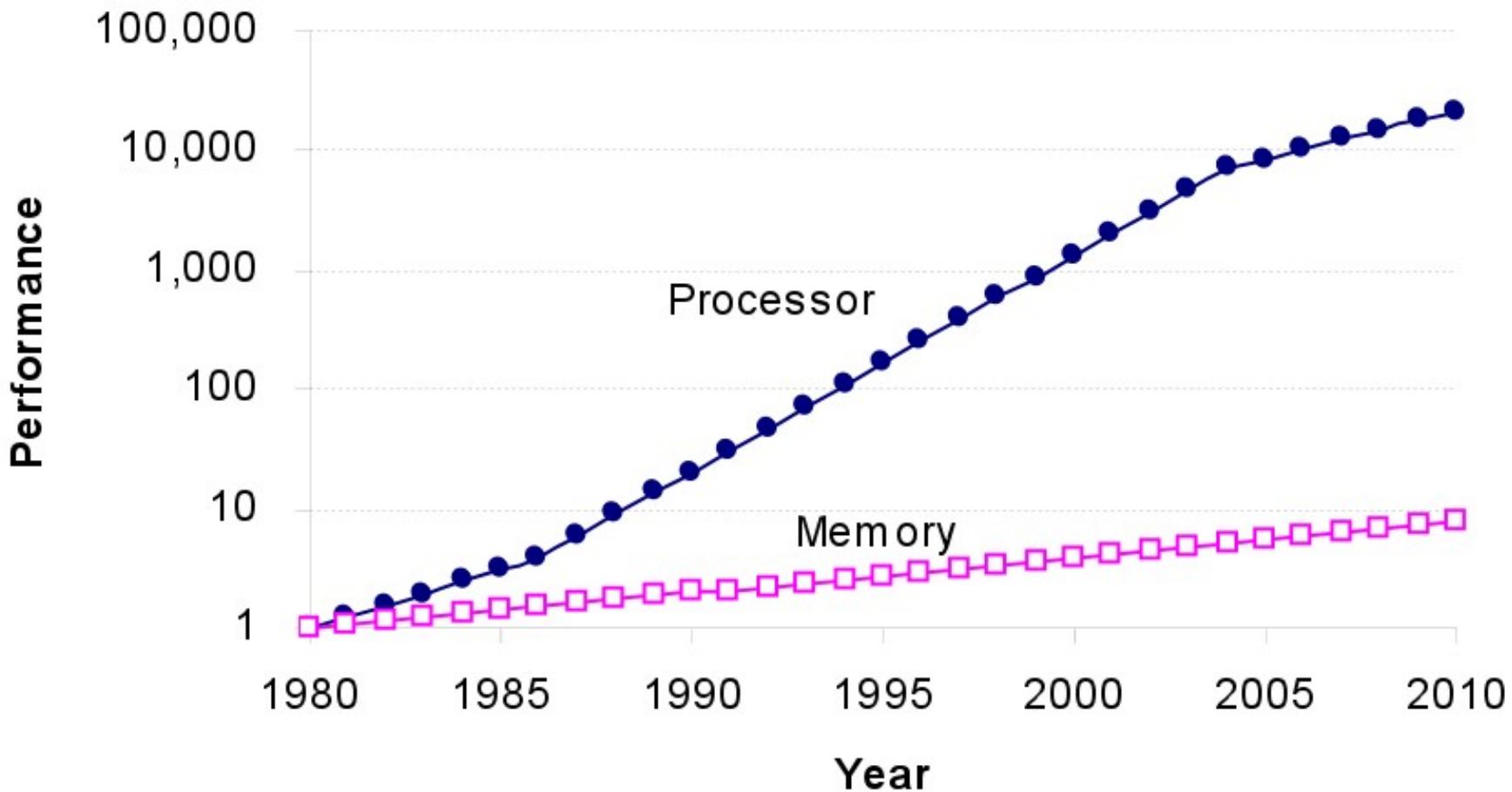
**Obrada** mnogo brža od **latencije** memorije potrebnog kapaciteta:

- 0.5 ns (zbrajanje 32b, P4@2GHz) vs. 50 ns (DDR2-800)
- za sada razmatramo **radnu** (glavnu) **memoriju**, RAM

**Propusnost u prosjeku** dovoljna (zbog sofisticiranih tehnika):

- potrebna propusnost:  $(\text{takt} / \text{CPI}) \times (1 + m) \times \text{riječ} \approx 4\text{GB/s}$ 
  - takt = 2 GHz, *prosječna riječ* = 3
  - m = 30% (učestalost memorijskih instrukcija)
    - 20% grananje, 50% aritmetika
  - CPI  $\in (1, 6)$ , uzmimo srednju vrijednost CPI=2 (P4, SPECint2000)
  - (CPI<sub>thmax</sub> (P4)=0.33)
- 4 GB/s (vidi gore) vs 6.4 GB/s (DDR2-800: 64b  $\times$  800 transfera/s)
- obratiti pažnju na to da ovdje nismo razmatrali grafiku!

# Nesrazmjer latencije memorije i slijedne performanse procesora i dalje raste...



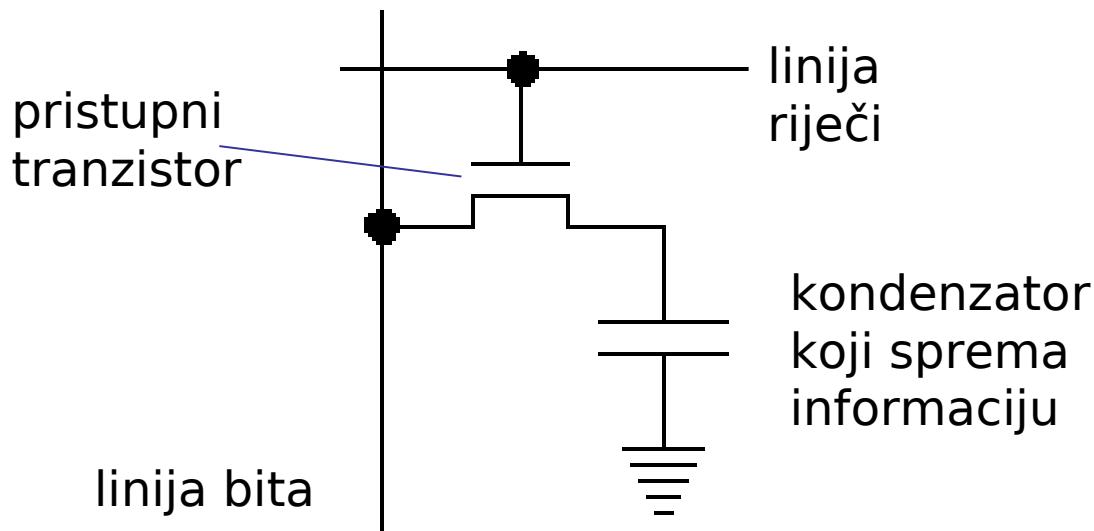
[Patterson08]

# 6. Priručne memorije

- 1. Svojstva i organizacija dinamičkog RAM-a**
2. Memorjska hijerarhija
3. Organizacija priručne memorije
4. Odabir parametara, performansa
5. Izvedbeni detalji

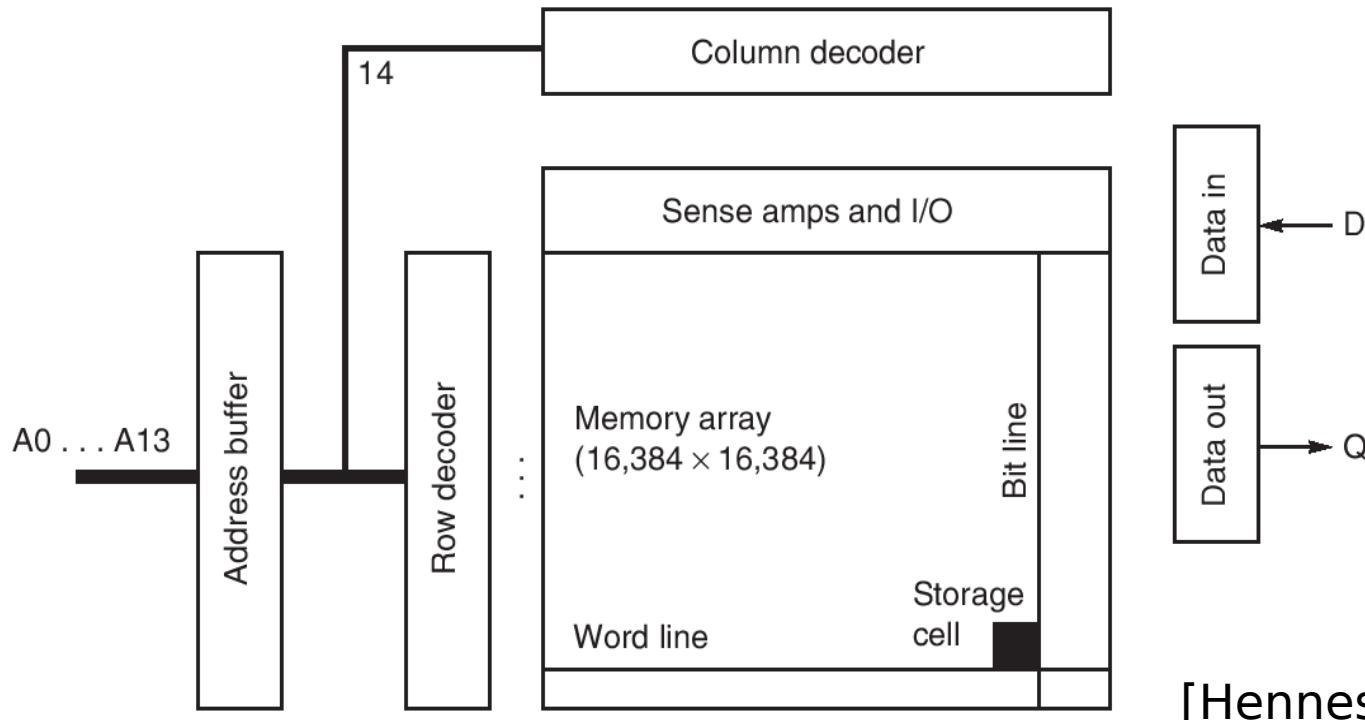
# Memorijske tehnologije (RAM)

- magnetni bubenjevi (1950)
- feritne jezgre (1960, pristup  $1\mu\text{s}$ )
- poluvodičke memorije (DRAM, 1970, Intel)
  - DRAM: tehnologija izbora za **glavnu** memoriju
  - veličina 1T ćelije DRAM-a odgovara veličini tranzistora
  - DRAM vs SRAM:  $10\times$  **gušći**,  $100\times$  **jeftiniji**,  $40\times$  **sporiji**



# Struktura DRAM memorije

- informacija smještena u kvadratnom polju 1T ćelija
  - optimalne veličine dekodera i duljine prospoa

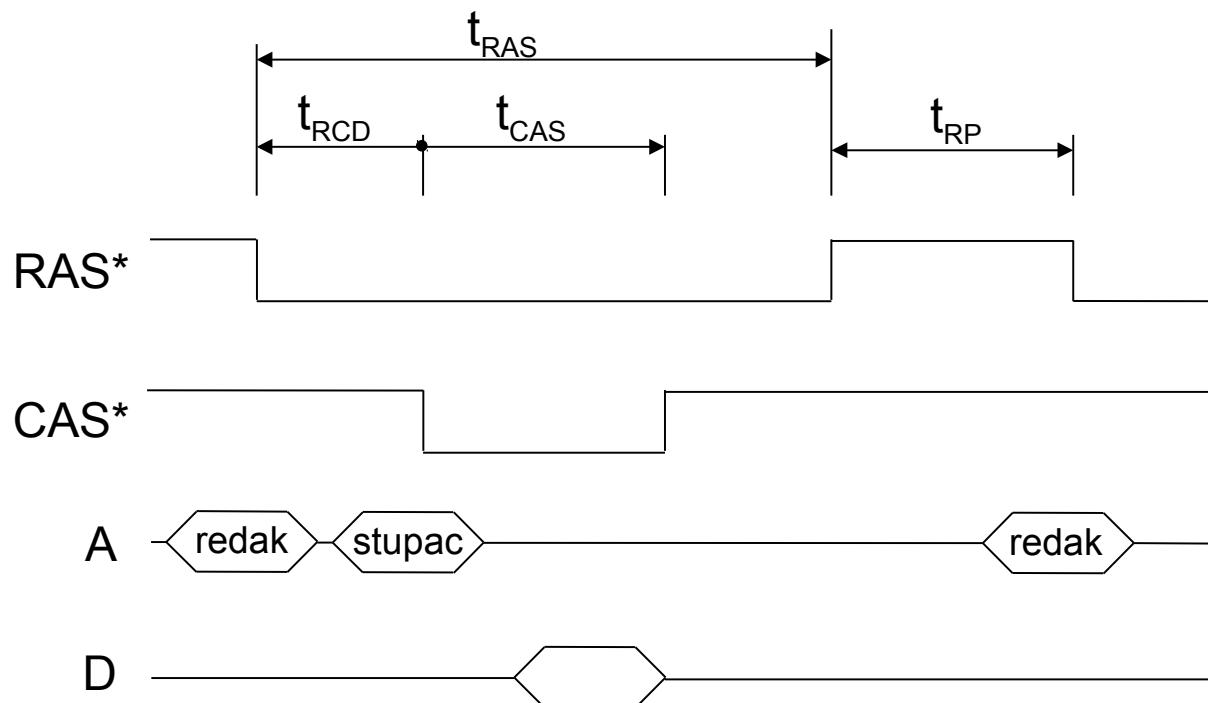


[Hennessy07]

# Vremenski dijagram pristupa DRAM-u

DRAM ciklus:

- aktiviranje retka (dekodiranje, pojačavanje i spremanje,  $t_{RCD}$ )
- pristup stupcu (odabir bitova retka, čitanje ili pisanje,  $t_{CAS}$ )
- prednabijanje linija bitova (potrebno prije novog aktiviranja,  $t_{RP}$ )



# Pristupi za povećanje propusnosti DRAM-a (1)

## 1. brzi pristup retku (fast page mode)

- redak ostaje selektiran, bitovi stupca izlaze u ritmu t\_CAS
- jeftin način brzog pristupa susjednim podatcima

## 2. paralelna organizacija memorijskog modula

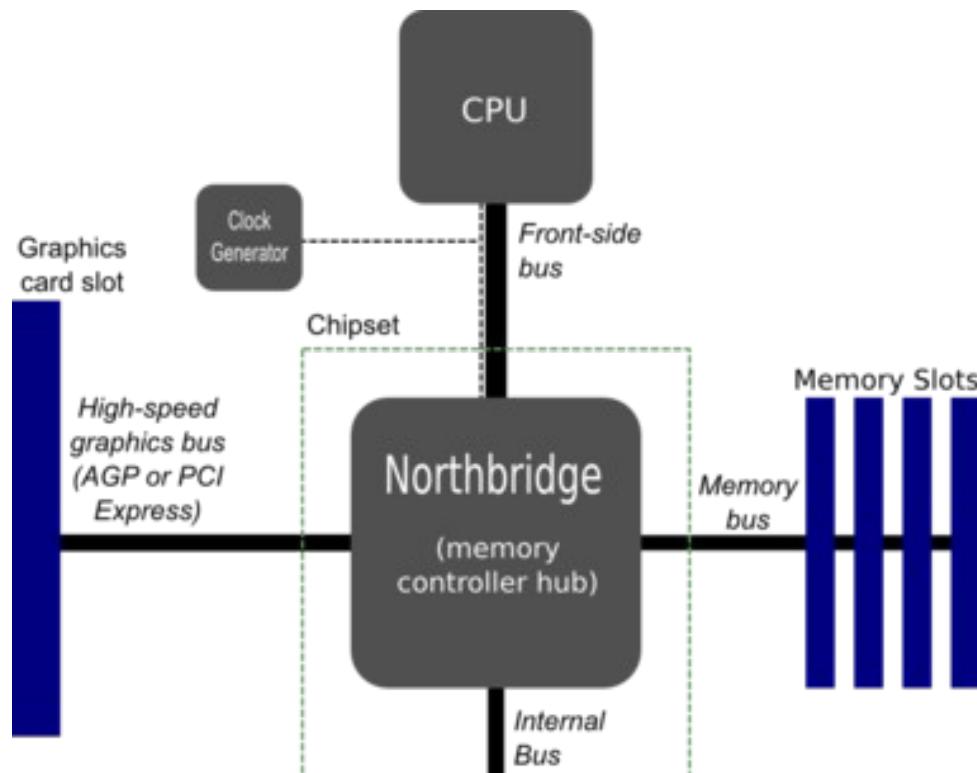
- oblikovanje memorijskog modula od više pojedinačnih sklopova
- svaki sklop odgovoran za smještanje dijela riječi
- usporedan pristup prednost, uz veću cijenu i potrošak energije
- tipični kompromis: 4 ili 8 bitova na svakom sklopu

# Pristupi za povećanje propusnosti DRAM-a (2)

## 3. sinkroni sabirnički protokol (SDRAM)

- preklapanje iščitavanja podataka i pristupa retku
  - prilikom svakog pristupa zapamtiti cijeli redak u dediciranom spremniku
  - tijekom prijenosa uzastopnih podataka iz dediciranog spremnika započeti pristup novom retku
  - sabirnica mora biti sinkrona (imati signal vremenskog vođenja)
- **protočni** protokol pristupa memoriji:
  - naredbe se izdaju prije dovršavanja prethodne operacije!
  - dok se pribavljenih  $N$  bitova u grupama upućuju na vanjsku sabirnicu, traje pristup novom retku/stupcu
- ostvaruje se brži **grupni** prijenos
  - podatkovna sabirnica širine 64 bita
  - grupni (burst) prijenos: više uzastopnih 64-bitnih podatka
  - pojedinačni prijenos jednako brz ili sporiji
  - prikladno za servisiranje priručnih memorija i DMA
- nedostatak je složeno upravljanje
  - posao **memorijskog pristupnog sklopa** (MCH, northbridge)
  - ako propusnost nije kritična, bolje koristiti asinkroni protokol

# Uloga memorijskog pristupnog sklopa (memory controller, northbridge)



[Wikipedia]

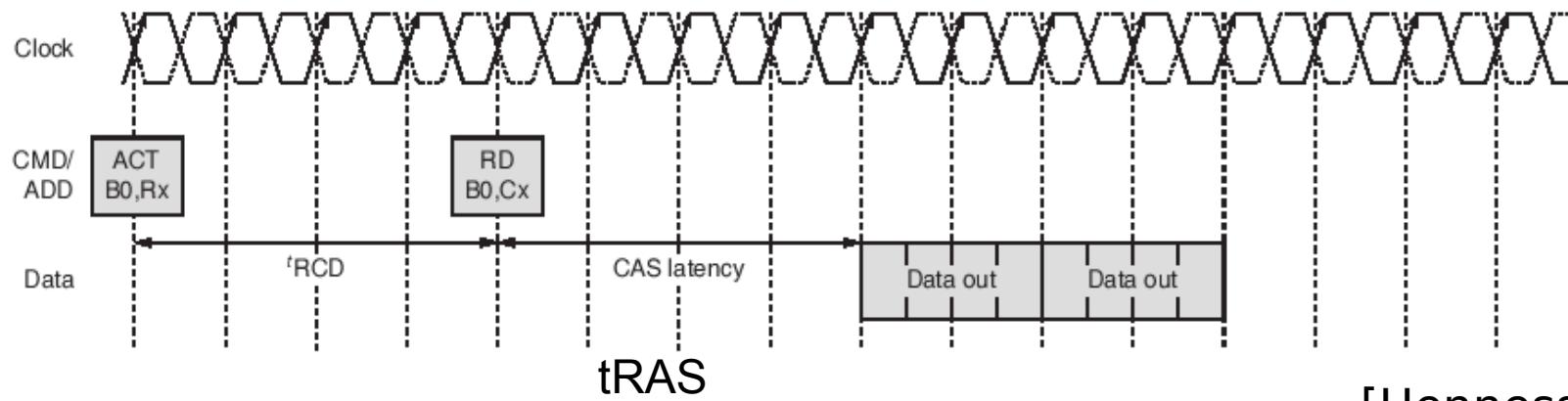
# Pristupi za povećanje propusnosti DRAM-a (3)

## 4. preplitanje (interleaving)

- ideja: smještati susjedne podatke u različite sklopove
  - i-ti logički podatak smjestiti u polje ( $i \bmod N$ )
  - usporedno adresirati svih  $N$  polja
  - pristigle podatke slati na vanjsku sabirnicu u ritmu  $t_{\text{CAS}}/N$
- preplitanje na razini pojedinačnog sklopa:
  - danas metoda izbora (DDR:  $N=2$ , DDR2:  $N=4$ , DDR3:  $N=8$ )
  - sklopovi sadrže više polja (bank) s prepletenim podatcima
  - npr (64 Mb): umjesto 1 polja  $8192 \times 8192$  imamo 4 polja  $4096 \times 4096$
  - propusnost na vanjskoj sabirnici modula  $N \times$  veća od memorijске frekvencije!
- preplitanje na razini matične ploče (dual channel):
  - najbolji rezultati uz odgovarajuće memorijске module
  - manje praktično, rjeđe korišteno

# Prepleteni sinkroni dinamički RAM (SDRAM DDR)

- SDRAM DDR2: industrijski standard ([www.jedec.org](http://www.jedec.org))
- preplitanje  $\times 4$ : min. 4 uzastopna čitanja/pisanja po 64 bita  
DDR2-800: DRAM 200MHz, BUS 400Mhz (DDR), FSB 800 MHz
- latencija po fazama izražena u ciklusima **vanjske** memorijске sabirnice (CAS-RCD-RP-RAS) na slici:  $t_{CAS}=4$ ,  $t_{RCD}=4$
- DDR2-800: tipično 5-5-5-15, odnosno 12.5ns-12.5ns-12.5ns
  - memorijска frekvencija iznosi 400 MHz  $\Rightarrow T = 2.5$  ns
  - ukupno vrijeme **slučajnog** pristupa:  $t = t_{RAS} + t_{RP} = (5+15)T = 50$  ns

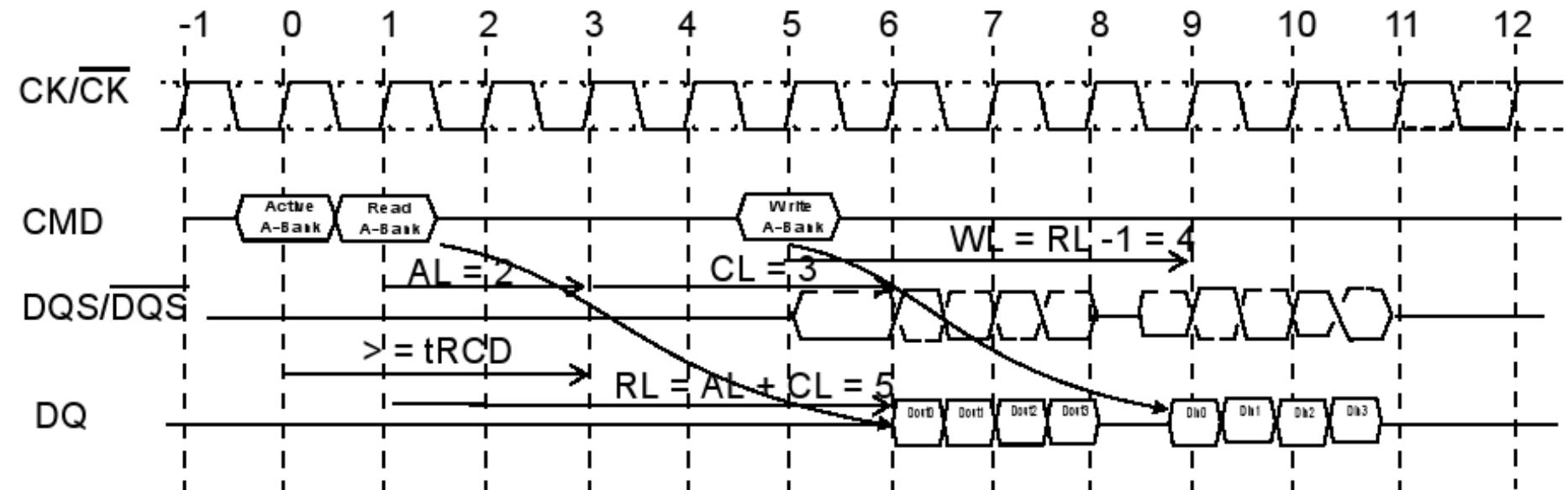


# Primjeri:

- npr DDR2-800:
  - vršna propusnost: 800 Mprijenos/a/s ( $6400\text{MB/s} \Rightarrow \text{PC-6400}$ )
  - istovremeno se prozivaju 4 polja na 200MHz
  - latencija sukladna memoriji DDR-400!
  - DIMM modul: 240 izvoda, tipično 8 sklopova po 8(+ECC) bita
- npr DDR3-X:
  - osmerostruko preplitanje
  - ukupna propusnost  $8\times$  veća od propusnosti pojedinačnog polja
  - latencija odgovara SDRAM memoriji na taktu X/8
- npr, DDR2-800 vs DDR3-800
  - koji modul ima bolju propusnost odnosno latenciju?

# SDRAM DDR2 standard (JEDEC):

- AL – additive latency
- CL – CAS latency
- RL,WL – read/write latency
- BL – burst length



[ $AL = 2$  and  $CL = 3$ ,  $RL = (AL + CL) = 5$ ,  $WL = (RL - 1) = 4$ ,  $BL = 4$ ]

# DRAM memorija, sažetak

- usko grlo performanse zbog velike latencije
  - 50 ns memorijske latencije **naprema** 0.5 ns takta CPU
  - više od 100 ciklusa latencije u najgorem slučaju!
  - u najboljem slučaju oko 25 ciklusa latencije  
(slijedni pristup, t\_CAS)
- sofisticiranom organizacijom (1-4) postiže se veća propusnost uz jednaku latenciju slučajnog pristupa
- svaka instrukcija referencira memoriju 1.3 puta
  - 1× dohvati instrukcije + 30% memorijskih instrukcija
  - da bismo podržali izdavanje instrukcije u svakom taktu, trebalo bi nam više od 100 memorijskih pristupa u svakom trenutku (!!)

# 6. Priručne memorije

1. Svojstva i organizacija dinamičkog RAM-a
2. **Memorijska hijerarhija**
3. Organizacija priručne memorije
4. Odabir parametara, performansa
5. Izvedbeni detalji

## Prostorna i vremenska **lokalnost** pristupa

- **vremenska lokalnost**: korištene lokacije će se vjerojatno koristiti i u budućnosti
- **prostorna lokalnost**: lokacije blizu korištenih lokacija će se vjerojatno također koristiti

## Lokalnost pristupa u praksi:

- programska memorija: petlje, potprogrami
- podatkovna memorija: lokalne varijable (stog), članovi objekta, polja, konstante

Korištenje lokalnosti pristupa je **velika ideja**

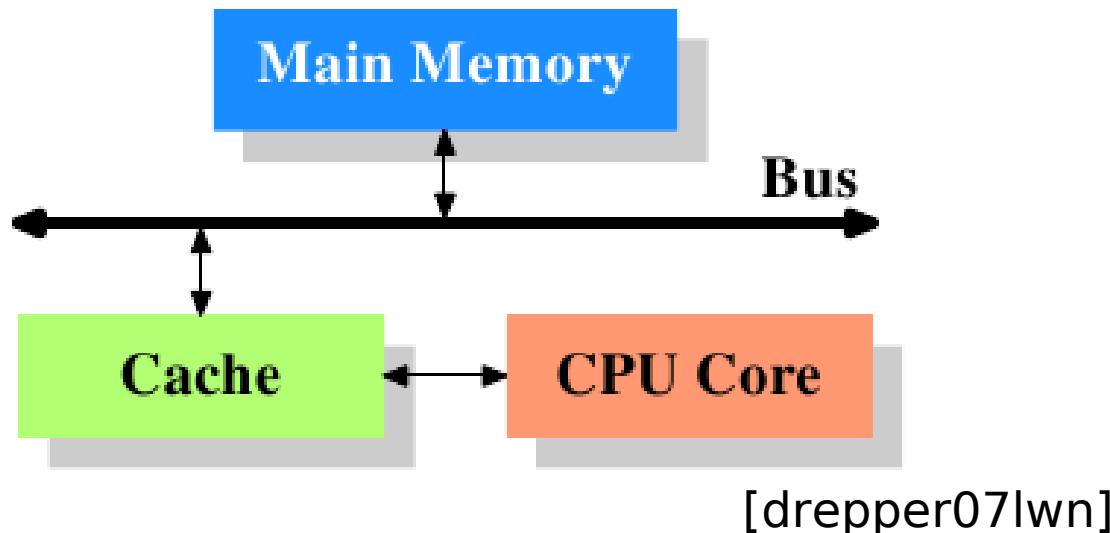
# Memorijska hijerarhija

- **ideja:** smanjiti prosječnu latenciju korištenjem lokalnih kopija “popularnih” podataka (radnog skupa) u bržoj memoriji
- niže razine imaju veći kapacitet, veću latenciju i manju cijenu
- prividno, računalo ima kapacitet diska, a brzinu registara

|  |  |
|--|--|
| zaporni sklopovi<br>protočne strukture | .05 ns ( $1/5 \times \Delta T_{CPU}$ ), 100B |
| registri (SRAM)                        | .25 ns ( $1 \times \Delta T_{CPU}$ ), 500B   |
| <b>L1 cache</b> (SRAM)                 | 1 ns ( $4 \times \Delta T_{CPU}$ ), 64kB     |
| RAM (DRAM)                             | 50 ns, 1 GB                                  |
| diskovi                                | 10 ms, 1 TB                                  |

# Osnove priručnih memorija (PM, cache)

- cache: mala brza memorija, blizu procesora
- kad se referencira podatak:
  - ako je kopija podatka u priručnoj memoriji, vrati nju
  - inače:
    - ako je potrebno, izbaci nešto iz priručne memorije
    - dohvati podatak iz glavne memorije (dohvati i susjede)



# Važna pitanja

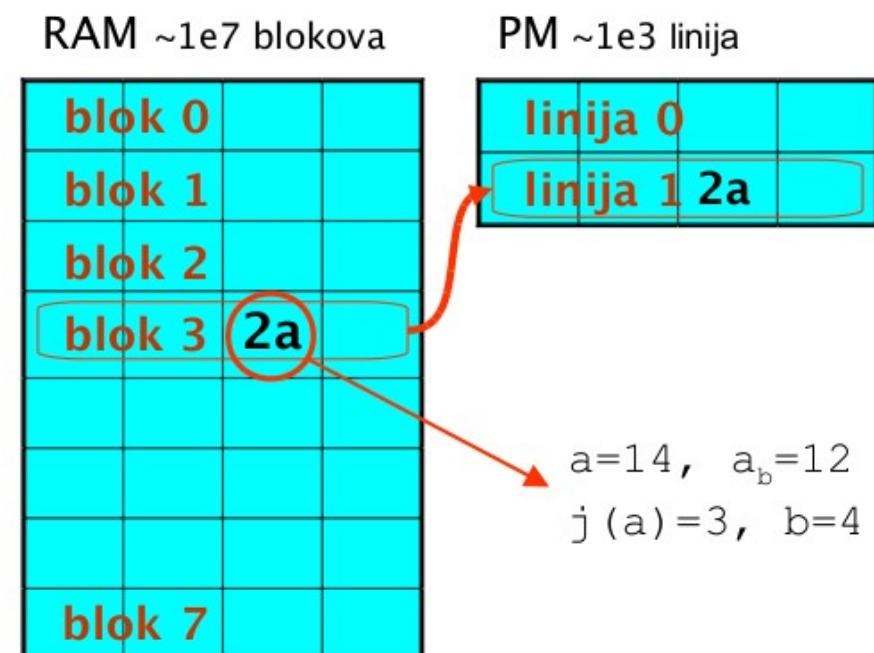
- kamo smjestiti koju memorijsku lokaciju?  
(RAM>PM  $\Rightarrow$  više lokacija RAM-a mora se moći preslikati u istu lokaciju cachea!)
- kako saznati da li je tražena adresa u priručnoj memoriji?
- kako brzo pristupiti cacheiranoj kopiji podatka?
- koje lokacije izbaciti van kad se javi potreba?
- kada upisati promijenjeni podatak natrag u glavnu memoriju?

# 6. Priručne memorije

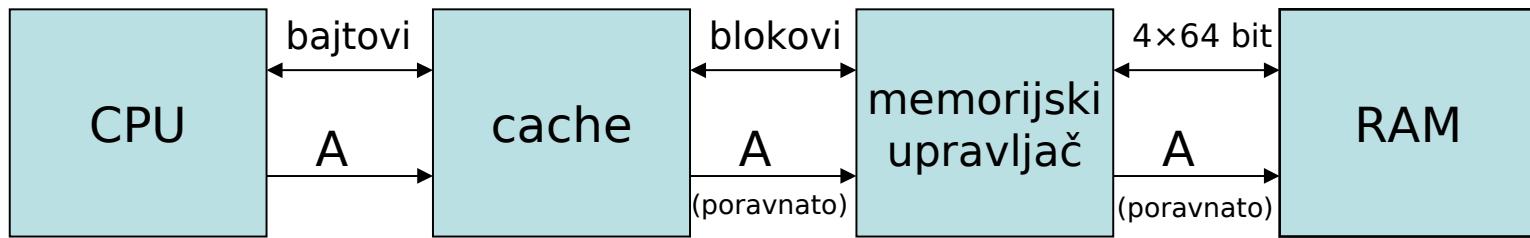
1. Svojstva i organizacija dinamičkog RAM-a
2. Memorjska hijerarhija
- 3. Organizacija priručne memorije**
4. Odabir parametara, performansa
5. Izvedbeni detalji

# Važna ideja: preslikavati poravnate blokove

- PM smješta kopije **poravnatih blokova** podataka iz RAM-a
  - veličina bloka **b** izražena u bajtovima je potencija broja 2
  - adresa bloka  $a_b$  poravnata u odnosu na veličinu bloka:  $a_b \bmod b = 0$
- Svaki bajt memorije pripada točno jednom bloku
  - adresa  $a$  pripada bloku s indeksom  $j(a) = \lfloor a/b \rfloor$  ("najveće cijelo")
  - adresa bloka koji sadrži  $a$  je  $a_b(a) = j(a) \cdot b$
  - npr ( $b=4$ ,  $a = 14$ ):
    - $j(a) = \lfloor a/b \rfloor = \lfloor 3.5 \rfloor = 3$ ,
    - $a_b(a) = j \cdot b = 3 \cdot 4 = 12$
  - prihvatno mjesto za tako poravnate blokove nazivamo **linijom** priručne memorije (cache line)

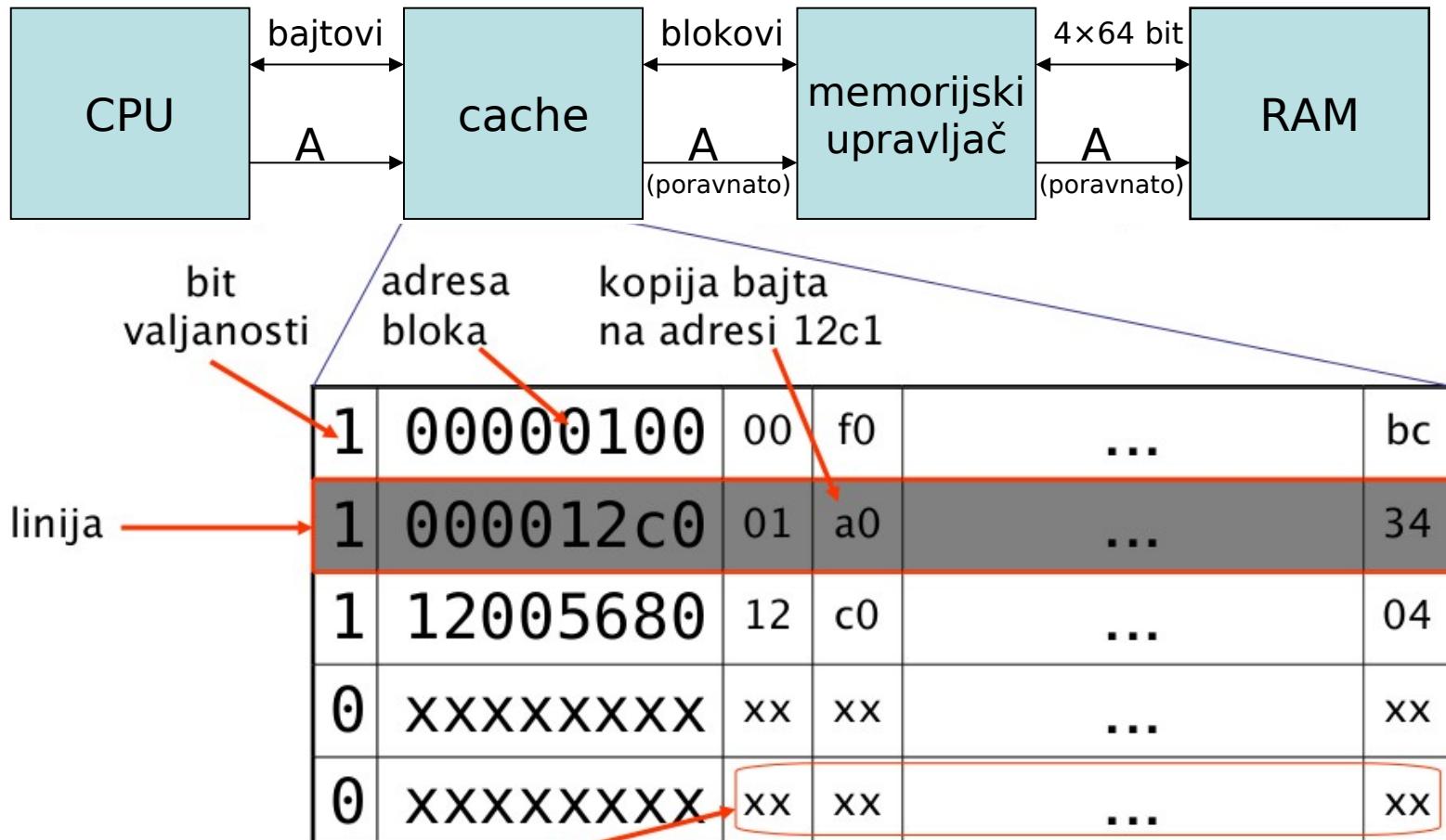


# Poravnati blokovi → efikasan transfer



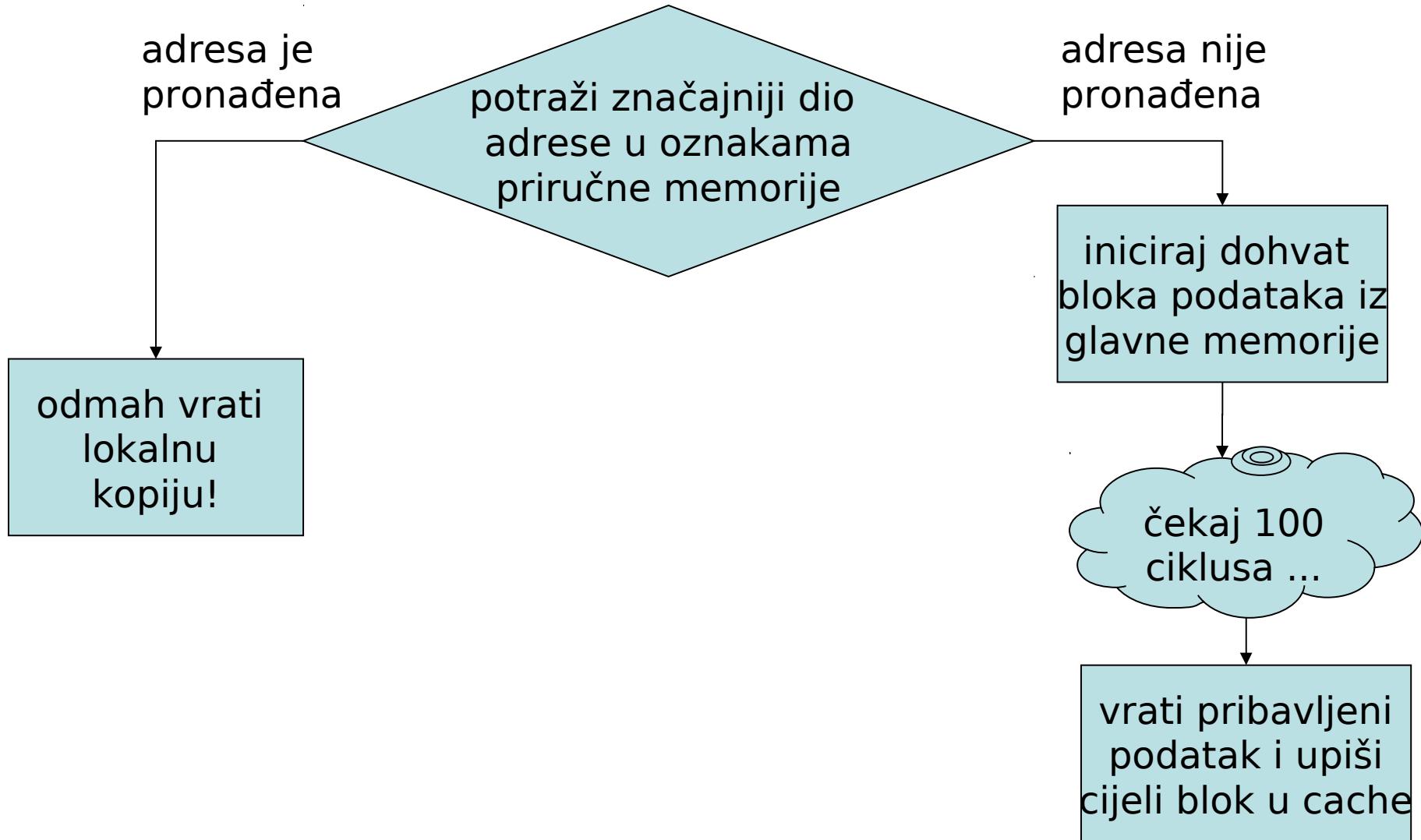
- preslikavanje poravnatih blokova osigurava:
  - brzi transfer iz glavne memorije (ili prema njoj)
  - brzi pristup cacheiranim podatcima (o tome ćemo pričati)
  - dobru prilagodbu prostornoj lokalnosti podataka
- osnovni parametri PM: broj linija  $n$  i veličina linije  $b$ 
  - kapacitet priručne memorije tada je:  $s = n \cdot b$
  - linija je kvant prijenosa iz glavne memorije u priručnu (time favoriziramo prostornu lokalnost)

# Najjednostavnija organizacija PM



poravnati **blok** podataka (veličina bloka **b** tipično 32B ili 64B)

# Čitanje priručne memorije



# Oblikovanje preslikavanja adresa → linija

- koji memorijski blokovi se mogu upisati u koju liniju?
- moguće izvedbe:
  - **izravno** preslikavanje  
(blok se preslikava u točno određenu liniju)
  - **potpuno asocijativno** preslikavanje  
(blok se može preslikati u bilo koju liniju)
  - **skupno asocijativno** preslikavanje  
(blok se može preslikati u neku od **a** linija)
    - npr, **a=4**: četveroelementno asocijativno preslikavanje  
(4-way associative mapping)
    - npr, **a=1**: izravno preslikavanje
    - npr, **a=n**: potpuno asocijativno preslikavanje

# Izvedba **izravnog** preslikavanja

- svaki memorijski blok može se preslikati u **samo jednu** liniju
- princip:** blokove linijama dodjeljivati po modulu **n**
- i**-ta linija PM prima blokove s rednim brojevima  $\{j \bmod n = i\}$
- olakšano provjeravanje i traženje: kopija može biti samo na jednom mjestu (najlakša implementacija)

glavna memorija

|    |        |
|----|--------|
| 00 | blok 0 |
| 04 | blok 1 |
| 08 | blok 2 |
| 0c | 2a     |
| 10 |        |
| 14 |        |
| 18 |        |
| 1c | blok 7 |

priručna memorija



Kad procesor traži bajt na adresi  $a = 0e$ , cache učita cijeli blok  $j(a)$  u liniju  $i(a)$

indeks bloka:  $j = [a/b] = 3$

adresa bloka:  $a_b = j \cdot b = 3 \cdot 4 = 0c$

linija:  $i = j \bmod n = 3 \bmod 2 = 1$

pomak:  $p = a \bmod b = 0e \bmod 4 = 2$

$a=0e$

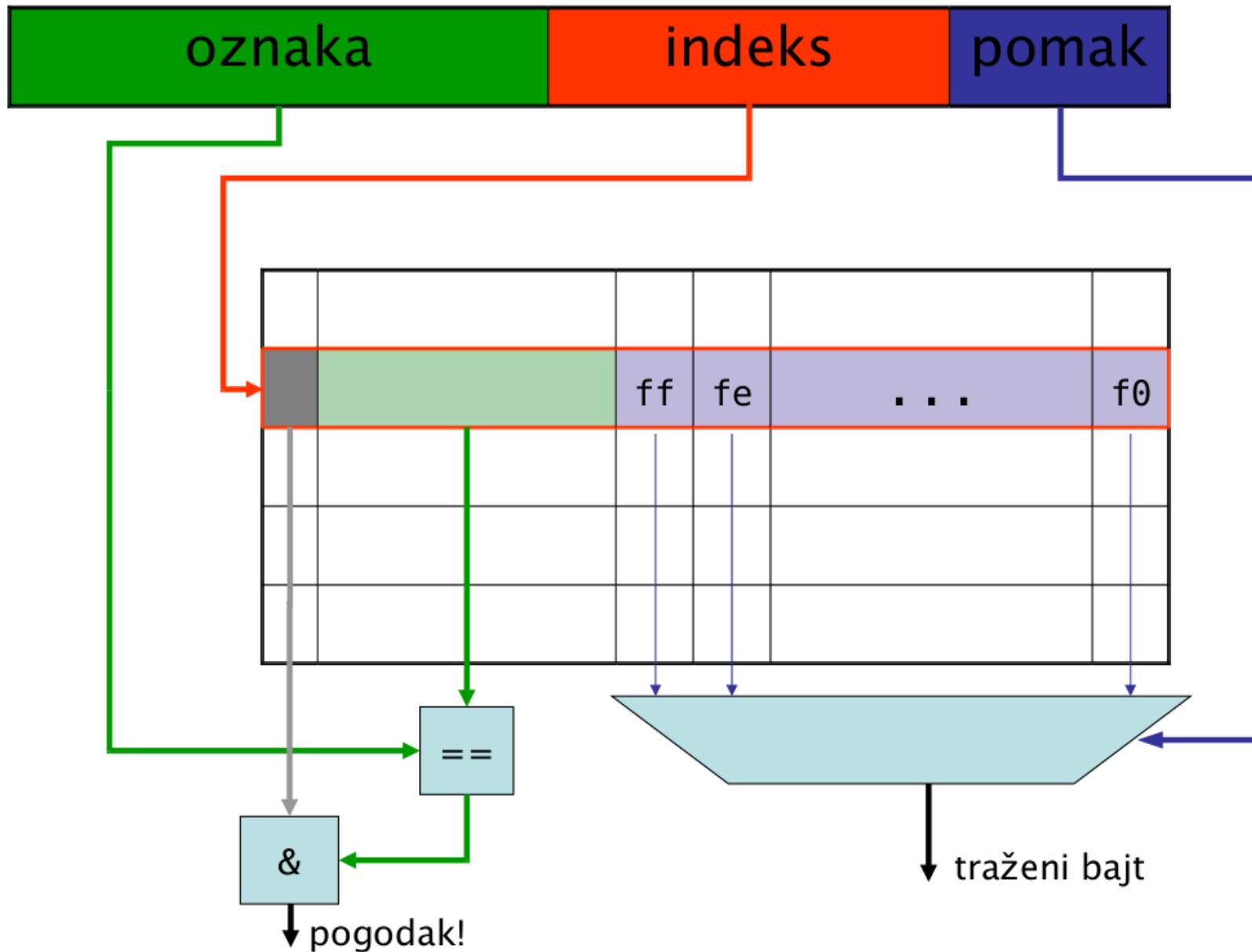
# Detalji izravnog preslikavanja

- ukoliko odaberemo “ligepe” **b** i **n**, implementacija je laka
- podijelimo bitove adrese koju generira procesor u tri polja:

$$a = \text{ooooooooooooooo} \text{iiiiiiii} \text{pppp}$$

- polje **pomaka** (offset)  $\mathbf{p}(a) = a \bmod \mathbf{b}$ 
  - adresira bajt u bloku (odnosno liniji)
- polje **indeksa** linije  $\mathbf{i}(a) = \mathbf{j}(a) \bmod \mathbf{n}$ ,  $\mathbf{j}(a) = \lfloor a / \mathbf{b} \rfloor$ 
  - određuje prihvatu liniju u priručnoj memoriji
- polje **oznake** (tag)  $\mathbf{o}(a) = \lfloor a / (\mathbf{n} \cdot \mathbf{b}) \rfloor$ 
  - koristi se za identifikaciju bloka u priručnoj memoriji
- vrijedi:  $\mathbf{b} = 2^{w(p)}$ ,  $\mathbf{n} = 2^{w(i)}$
- konkretni primjer je za cache s izravnim preslikavanjem od 1024 linije po 16B (16kB)

## PM s izravnim preslikavanjem



**Zadatak:** neka je zadano:

- PM s izravnim preslikavanjem  $s=16kB$ ,  $b=16B$ , 32b adresa
- slijed pristupa: 0x00000014, 0x0000001C, 0x00000034, 0x00008014

**Odrediti** koji će pristupi rezultirati pogotkom!

- pretpostaviti bajtne pristupe početno praznom cacheu

**Struktura adrese:**

- $w(p) = \log_2 b = 4$
- $n = s/b = 16kB/16B = 1024$
- $w(i) = \log_2(n) = 10$ ;  $w(o) = 32 - 14 = 18$
- **adresa:** 

**Analiza pristupa:**

- 0x00000014 : linija 1, pomak 4 (promašaj)
- 0x0000001C : linija 1, pomak C (pogodak)
- 0x00000034 : linija 3, pomak 4 (promašaj)
- 0x00008014 : linija 1, pomak 4 (promašaj s promjenom)

0x00000030?

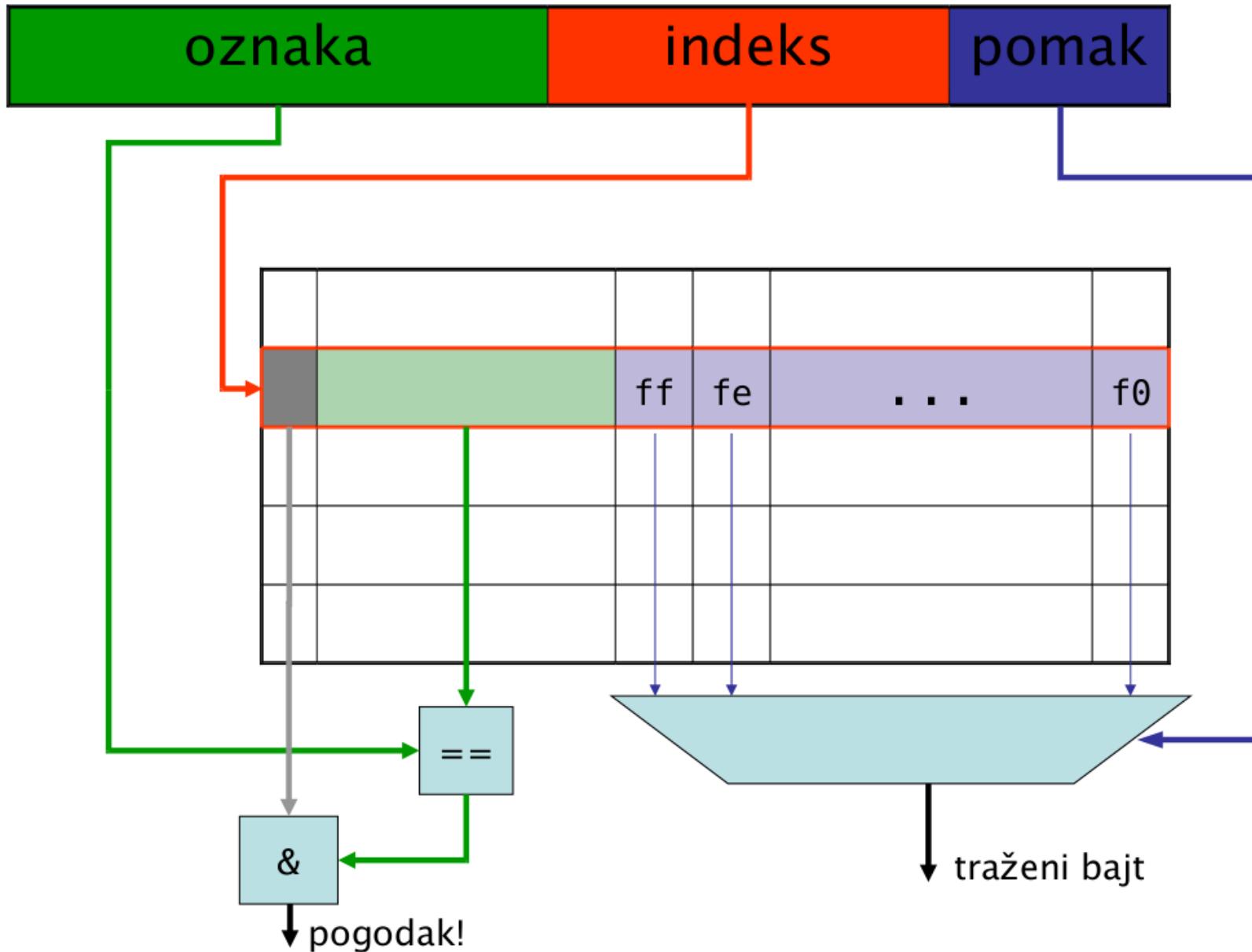
0x0000001C?

0x00008020?

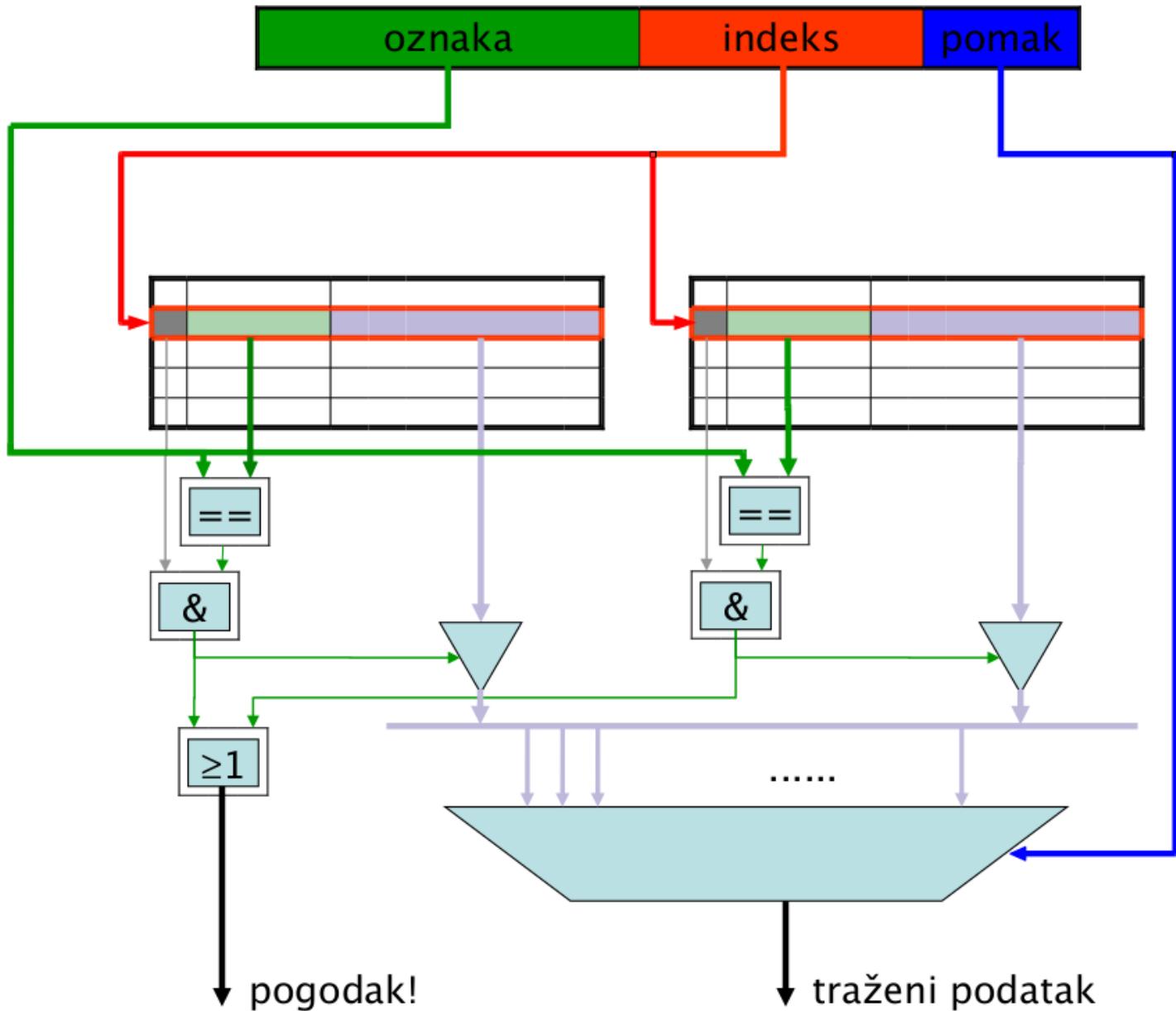
# Asocijativno preslikavanje

- Izravno preslikavanje je lako izvesti ali ima nedostataka
  - pristupi s istim indeksom problematični ( $0x00080014 \rightarrow 0x00000001c$ )
  - opetovano pražnjenje i punjenje iste linije
- lijek: priručne memorije sa skupnom asocijativnošću **a**
  - indeks sada adresira skup od **a** linija priručne memorije!
  - odabir između **a** adresiranih linija na temelju **oznake**
  - u odnosu na izravno preslikavanje, indeks je uži:  $w(i) = \log_2(n/a)$ !
  - povećanje asocijativnosti usporava cache (jednostavnije strukture brže!)
- npr, 32-bitna adresa, 4kB cache, linija od 64B:
  - ukupno **n=64** linije (4096B/64B) po **b=64** bajtova
  - izravno preslikavanje: pomak(6b), indeks(6), oznaka(20)
  - 2× asocijativno preslikavanje: pomak(6b), indeks(5), oznaka(21)
  - 4× asocijativno preslikavanje: pomak(6b), indeks(4), oznaka(22)
  - potpuno asocijativno preslikavanje: pomak(6b), indeks(0), oznaka(26)

## PM s izravnim preslikavanjem

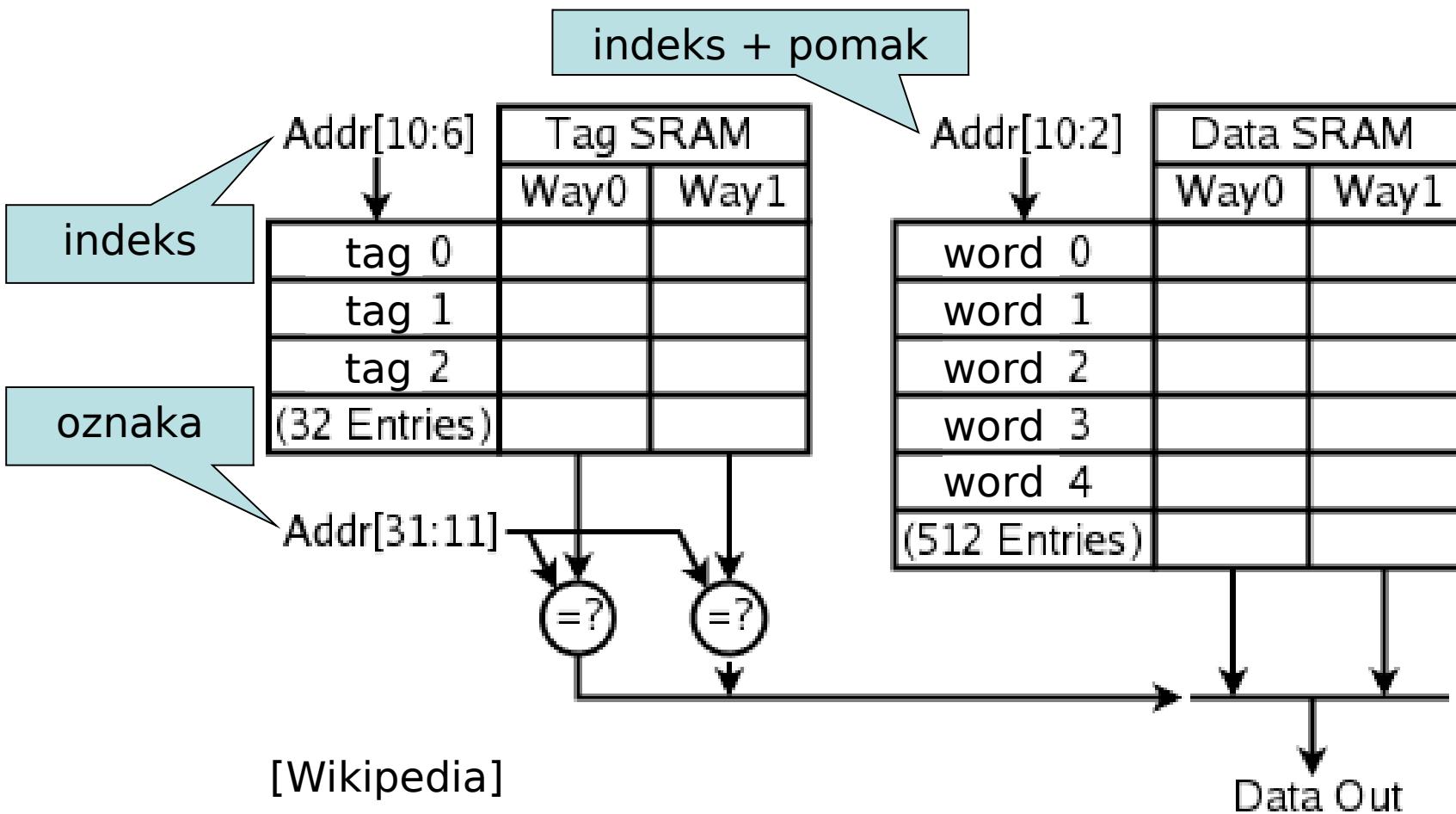


## Asocijativno preslikavanje, a=2



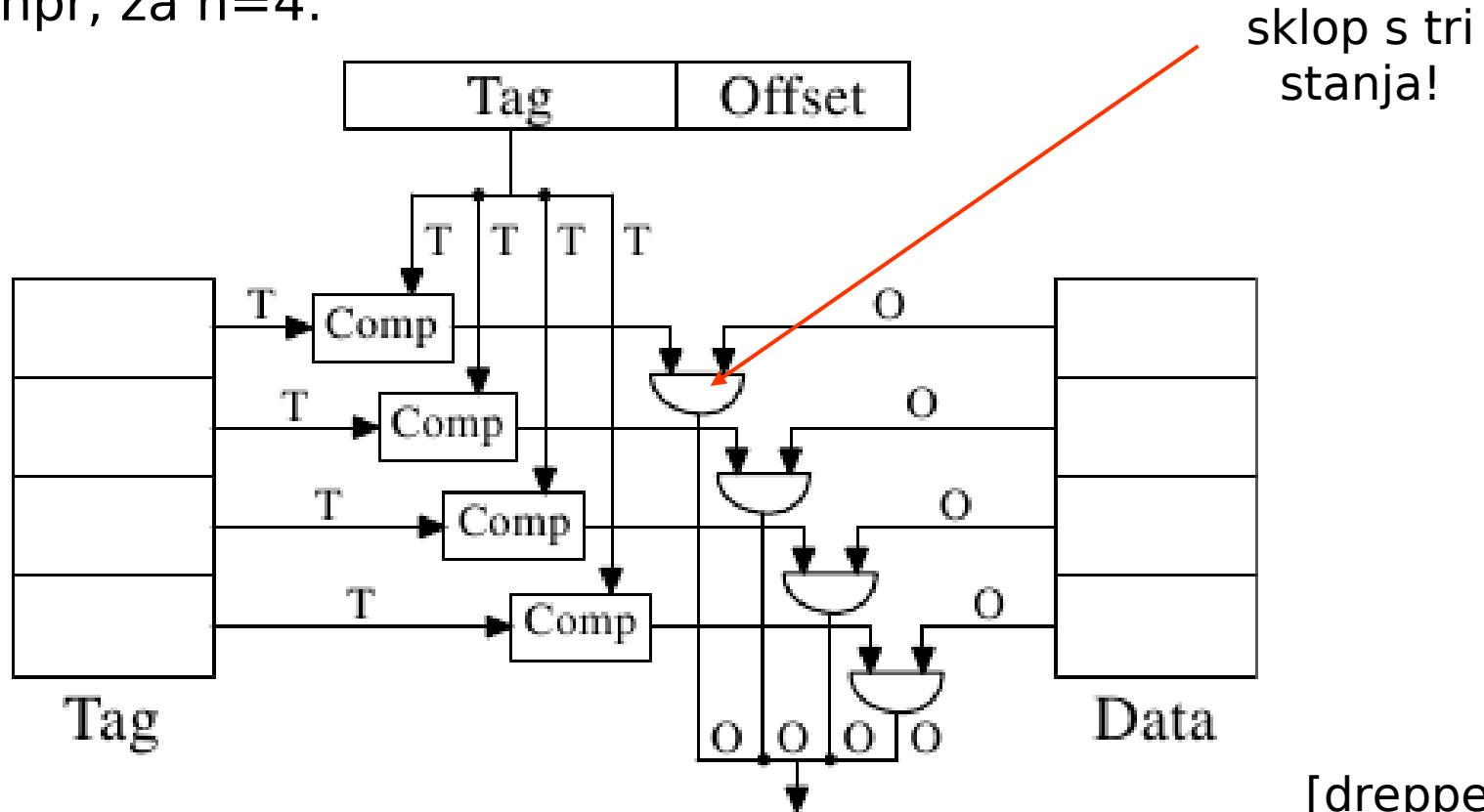
4kB cache, 2x asocijativan, 64-bajtna linija, 32-bitni pristup

- 4 bita za **pomak** 32-bitne riječi, 5 bitova za **indeks skupa**, 21 bit za **oznaku** adrese (ukupno 30 korisnih bitova adrese)
- u izvedbi, odvaja se **identifikacijski** dio od **podatkovnog** dijela PM
- identifikacijski dio (tablicu oznaka) izravno adresira polje indeksa
- podatkovni dio adresiraju kombinacija indeksa skupa i pomaka ( $5+4=9$  bitova)
- eventualni odabir pribavljenih riječi obavljaju izlazni sklopovi s tri stanja



# Potpuno asocijativno preslikavanje

- oznake **svih** linija PM uspoređuju se istovremeno!
- nema polja indeksa: **a=n**,  $w(i)=0$
- samo za male PM s vrlo skupim promašajima
- npr, za  $n=4$ :



**Zadano:** 2× asocijativna PM s bajtnom zrnatošću, kapacitet 64 kB, linija od 64 B, 32-bitna adresa, 2 servisna bita (**Valid**,**Dirty**).

**Odrediti** ukupni broj bitova linije PM.

**Struktura adrese:**

- $w(p) = \log_2 b = 6$
- $n = s/b = 64kB/64B = 1024$
- $w(i) = \log_2(n/a) = 9;$
- $w(o) = 32 - 9 - 6 = 17$
- **adresa:** 0000000000000000iiiiiiiii pppppp

**Struktura linije:**

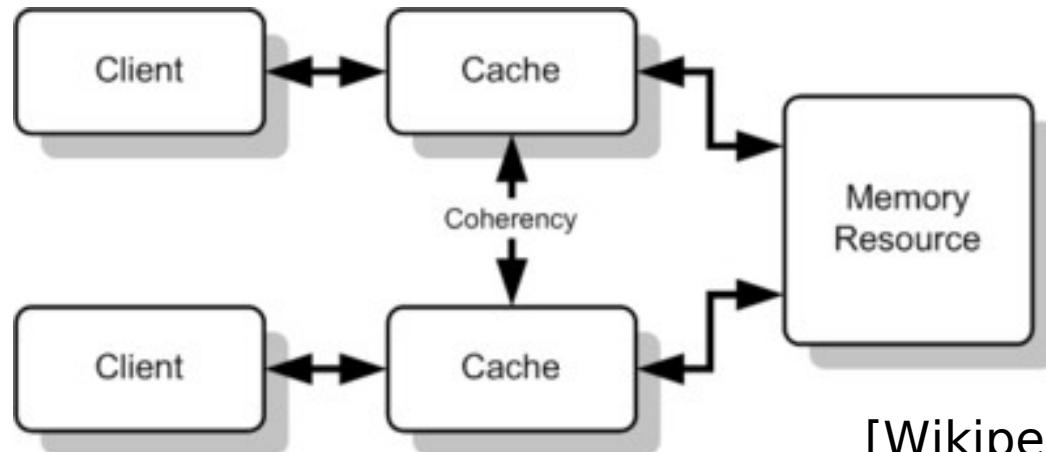
- oznaka (17b), servisni bitovi (2b), podatci (64B)
- ukupno:  $17 + 2 + 64 * 8 = 531b$

# Organizacija priručne memorije, **medusažetak**

- koristi se **lokalnost** pristupa kako bi se maksimirala performansa u **čestim** slučajevima
- transparentni prijenos blokova RAM-a u linije PM
- preslikavanje indeksiranjem ili asociranjem (izravno, višeelementno ili potpuno asocijativno)
- još nismo rekli:
  - kada upisati promijenjenu kopiju natrag u glavnu memoriju?
  - koje lokacije izbaciti van kad se javi potreba?
  - kako dimenzionirati cache (**n**, **b**, **a**)?

# Što napraviti nakon pisanja u priručnu memoriju?

1. novu vrijednost odmah proslijediti u memoriju (write-through)
  - najsigurniji pristup, ali najveći pritisak na memoriju
1. odgoditi upis (write-back)
  - upisati novu vrijednost samo u cache
  - dodati bit promjene ('dirty' bit) koji pamti da je kopija promijenjena
  - OS upisuje promijenjene linije pri promjeni konteksta ili UI operaciji
  - koherencija je osjetljivo pitanje, posebno kod MP sustava!



[Wikipedia]

## Algoritmi zamjene blokova

- kod izravnog preslikavanja, sve je **jasno**:
  - novi blok se upisuje na jedino mjesto, prethodni stanar se izbacuje (ako ga ima)
- kod skupno asocijativnih memorija moramo odabratи blok za izbacivanje
  - LRU: blok koji je najdavnije korišten se izbacuje
    - relativno dobri rezultati (vremenska lokalnost)
    - skupa implementacija za više od  $2 \times$  asocijativnost
  - NMRU: izbacuje se blok koji nije posljednji korišten
    - jeftina aproksimacija LRU
    - (pseudo) slučajan odabir izbačenog bloka
  - FIFO, random, ...

**Zadatak:** odrediti pogotke cachea u sustavu s 4-bitnim adresama:

- parametri cachea:  $a=2$ ,  $n=4$ ,  $b=1$ , LRU
- pristupi: 0, 2, 0, 1, 4, 0, 2, 3, 5, 4

**Struktura adrese:**

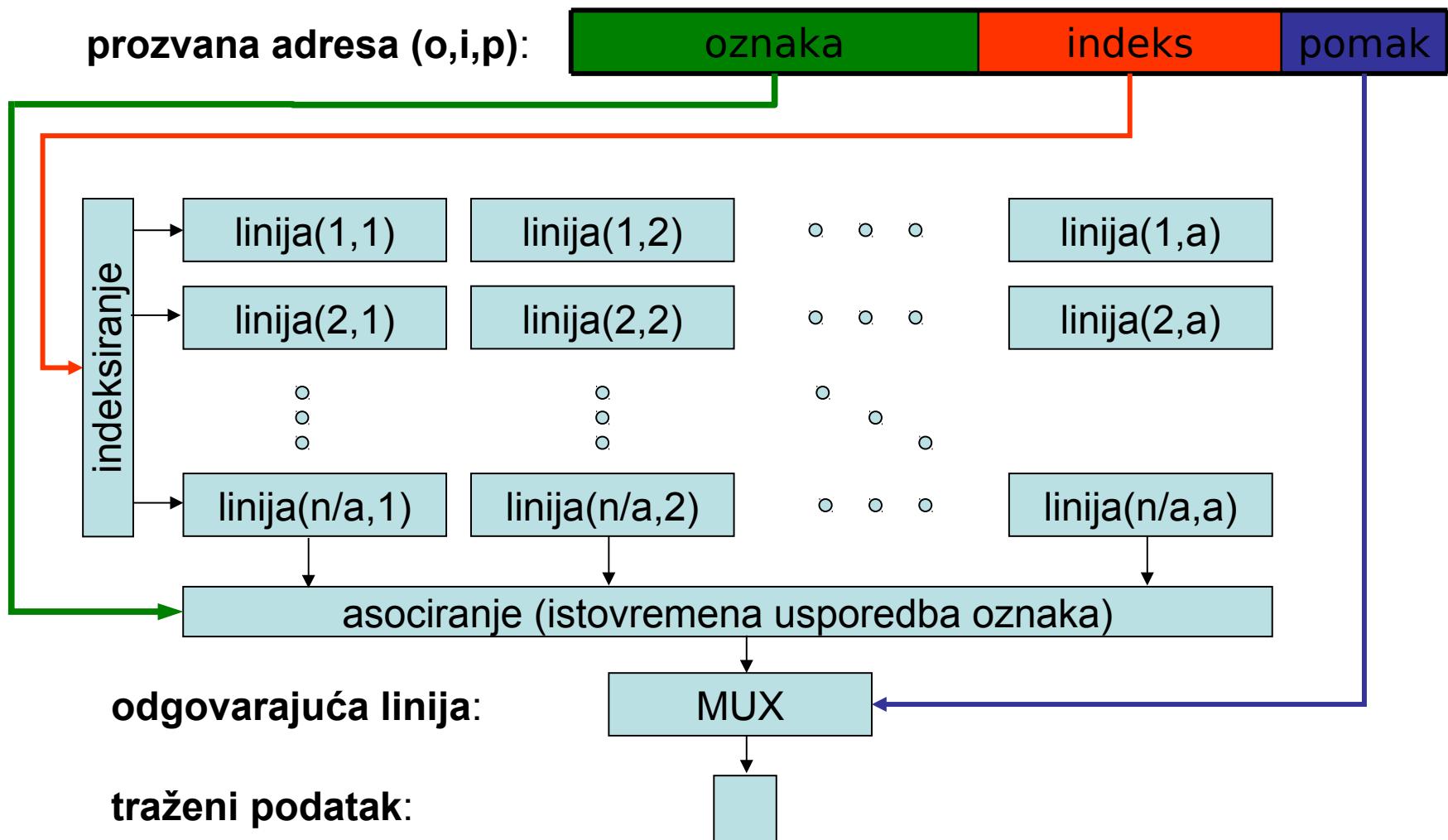
- adresa: **000i**
- dvije "linije" za parne, te dvije linije za neparne adrese

**Rješenje:**

- 0: promašaj
- 2: promašaj
- 0: pogodak
- 1: promašaj
- 4: promašaj (izbacuje 2)
- 0: pogodak
- 2: promašaj (izbacuje 4)
- 3: promašaj
- 5: promašaj (izbacuje 1)
- 4: promašaj (izbacuje 0)

## Sažetak parametara organizacije PM (**n**, **b**, **a**):

- broj linija **n**, broj bajta po liniji **b**, asocijativnost **a**, veličina **s = n × b**
- indeksiranjem odabiremo 1 od **n/a** skupova linija
- asociranjem odabiremo 1 od **a** linija u skupu

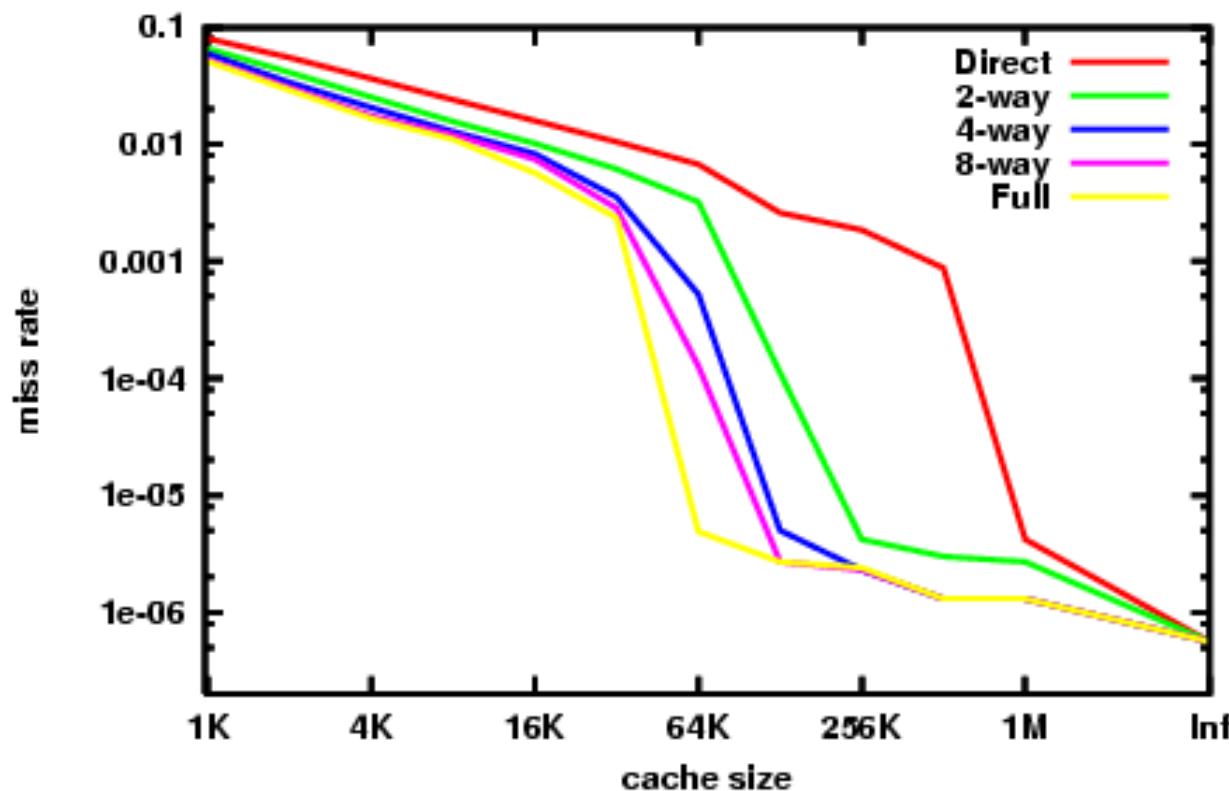


# 6. Priručne memorije

1. Svojstva i organizacija dinamičkog RAM-a
2. Memorijска hijerarhija
3. Organizacija priručne memorije
- 4. Odabir parametara, performansa**
5. Izvedbeni detalji

## Kako odabratи величину cachea i асocijativnost (n, a)?

- u mnogome ovisi o raspoloživoj tehnologiji
  - ključni su eksperimentalni podatci pribavljeni analizom izvođenja reprezentativnih programa (podatci na slici za SPECint2000)



## Analiza promašaja (3C)

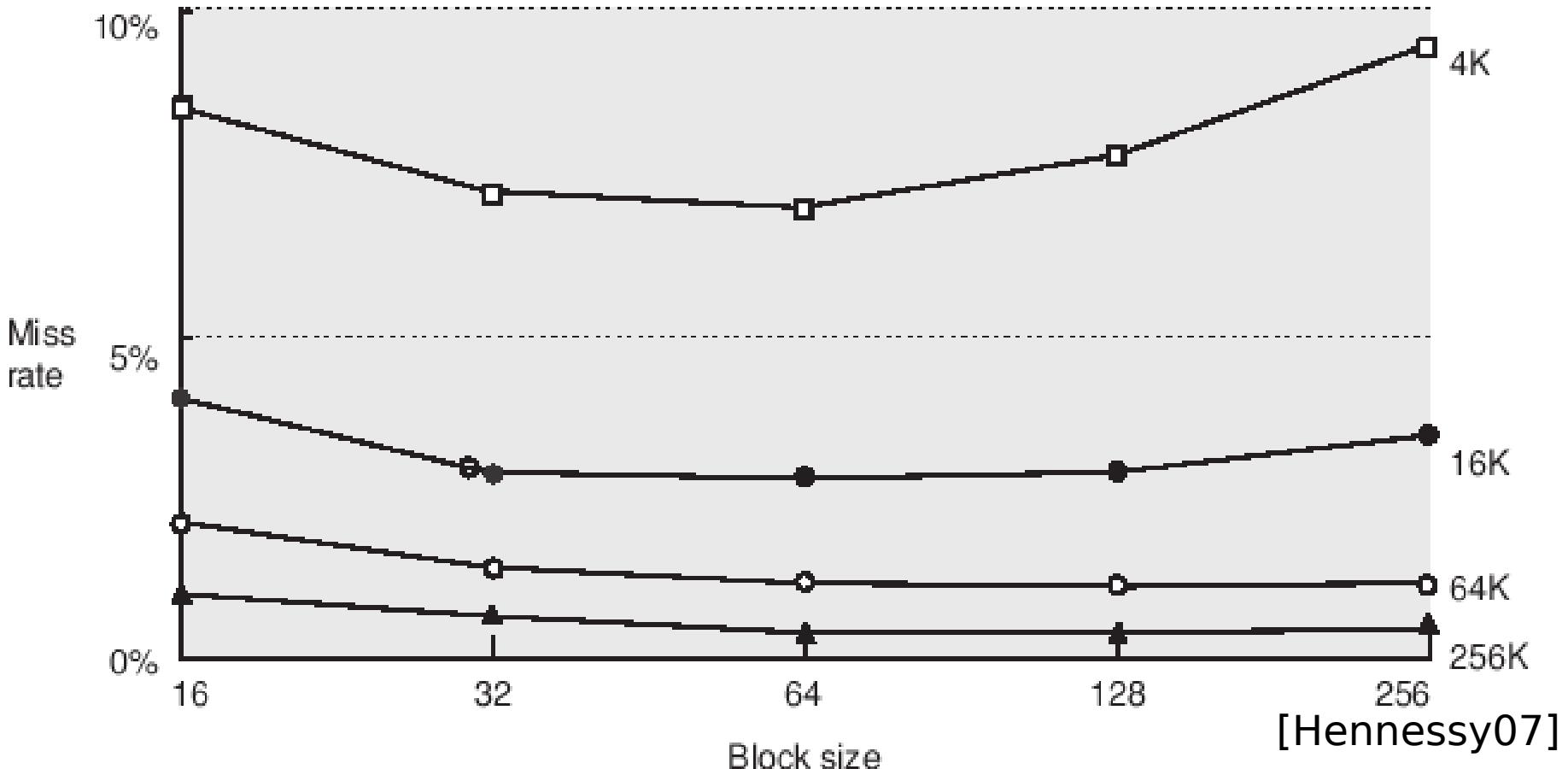
- **compulsory misses** (nezaobilazni):
  - učitavanje svih podataka koji su potrebni za izvođenje programa
  - ne ovise o veličini priručne memorije
  - desni dio grafa pokazuje koliko ih ima (oko 1e-6)
- **capacity misses** (zbog ograničenog broja linija):
  - dobar pokazatelj je graf za potpuno asocijativno preslikavanje
  - graf pokazuje da je radni skup (working set) između 32kB i 64kB
- **conflict misses** (zbog neidealne organizacije):
  - mogu se podijeliti na promašaje uslijed **ograničenog preslikavanja** i neprikladnog **algoritma zamjene**
  - pokazatelj promašaja uslijed ograničenog preslikavanja je usporedba s potpuno asocijativnim preslikavanjem
- Što smo naučili?
  - za velike i male priručne memorije, izravno preslikavanje prihvataljiv izbor (ali u višeprogramskom kontekstu višestruka asocijativnost je dobra ideja)
  - za srednje priručne memorije povećanje asocijativnosti nužna
  - veličina L1 cachea 32kB – 64kB
  - nema smisla imati priručnu memoriju veću od 1M (L2, L3, po korisniku)

# Kako odabratи veličinu linije?

- linija ne smije biti premala...
  - ne iskorištavamo prostornu lokalnost
  - važno za slijedne pristupe (instrukcije, elementi polja)
- ... ali ni prevelika
  - veća cijena promašaja (tražili 1B, dobili prijenos 64B)
  - manje linija  $\Rightarrow$  više promašaja, pogotovo u malom cacheu
- recept?
  - optimirati vrijeme pristupa prema modelu
  - najjednostavniji model: prosječno vrijeme pristupa memoriji
    - AMAT: average memory access time
    - model nije prikladan za procesore s dinamičkim raspoređivanjem!
  - $t_{AVG} = t(\text{pogodak}) \cdot v(\text{pogodak}) + t(\text{promašaj}) \cdot v(\text{promašaj})$
  - $t_{AVG} \approx t(\text{pogodak}) + t(\text{promašaj}) \cdot v(\text{promašaj})$
  - u praksi, 16B, 32B i 64B su najčešći odabiri

## Kako odabratи veličinu linije (2)?

- Ovisnost postotka promašaja za programe iz SPECint92:
  - najbolji rezultati za linije od 64B
  - za velike linije, učestalost promašaja raste
  - maksimum se pomiče udesno za velike priručne memorije
  - pažnja, velika PM  $\Rightarrow$  manji  $v$ (promašaj), ali i veći  $t$ (pogodak)!



# Prosječno vrijeme pristupa (primjer)

- Average memory access time (AMAT)
  - $t_{AVG} = t(\text{pogodak}) + v(\text{promašaj}) \times \text{cijena\_promašaja}$
  - cijena promašaja (engl. miss penalty) odgovara vremenu pristupa sljedećoj razini memorejske hijerarhije (L2, L3, RAM, disk)
- Zadano:
  - Period procesorskog takta  $T=1\text{ns}$
  - $t(\text{pogodak}) = 1\text{ T}$  (poznato, PM prati procesor)
  - $\text{cijena\_promašaja} = 100\text{ T}$  (tipična vrijednost, ovisi o memoriji)
  - $v(\text{promašaj}) = 1\%$  (izmjereno)
- $t_{AVG} = 1\text{ T} + 0.01 \times 100\text{ T} = 2\text{ T}$ 
  - efektivno vrijeme pristupa je dva ciklusa

## **Parametri** priručne memorije, sažetak:

- veličina memorije **s** (kompromis u odnosu na brzinu!)
- veličina linije **b**, broj linija **n=s/b**
- asocijativnost **a**
- algoritam zamjene (za **a>1**)
- strategija upisa (wb, **wt**)
- L2, L3 (u nastavku predavanja)?

## **Donekle** objektivan pristup dimenzioniranju:

- optimizacija prosječnog vremena pristupa
- čimbenici: budžet, tehnologija (veličina i složenost vs brzina), ciljani programi, ...

# Utjecaj PM na performansu računala

- Komponente procesorskog vremena:
  - normalno izvođenje programa
    - uključujući pogotke cachea
  - memorijski zastoji
    - uglavnom uslijed promašaja PM
- prosječni memorijski zastoj po instrukciji:
  - $T_{MZ} = v(\text{promašaja}) \times \text{cijena\_promašaja}$
- Utjecaj na CPI:

$$\text{CPI} = \text{CPI}_{\text{OSNOVNI}} + T_{MZ} / T_{\text{CPU}}$$

# Utjecaj PM na performansu (primjer)

- Zadano
  - promašaji instrukcijske PM:  $\nu(\text{PIPM}) = 2\%$
  - promašaji podatkovne PM:  $\nu(\text{PPPM}) = 4\%$
  - cijena promašaja:  $c = 100$  ciklusa
  - osnovni CPI (idealna PM):  $\text{CPI}_o = 2$
  - učestalost memorijskih instrukcija:  $\nu(\text{MI}) = 36\%$
- Broj ciklusa zastoja uslijed promašaja, po instrukciji:
  - instrukcijska PM:  $\text{CZPI}_{\text{IPM}} = 1 \times 0.02 \times 100 = 2$
  - podatkovna PM:  $\text{CZPI}_{\text{PPM}} = 0.36 \times 0.04 \times 100 = 1.44$
- Stvarni CPI =  $2 + 2 + 1.44 = 5.44$ 
  - usporenje uslijed neidealne PM =  $2.72 \times !$

# Utjecaj PM na performansu, sažetak

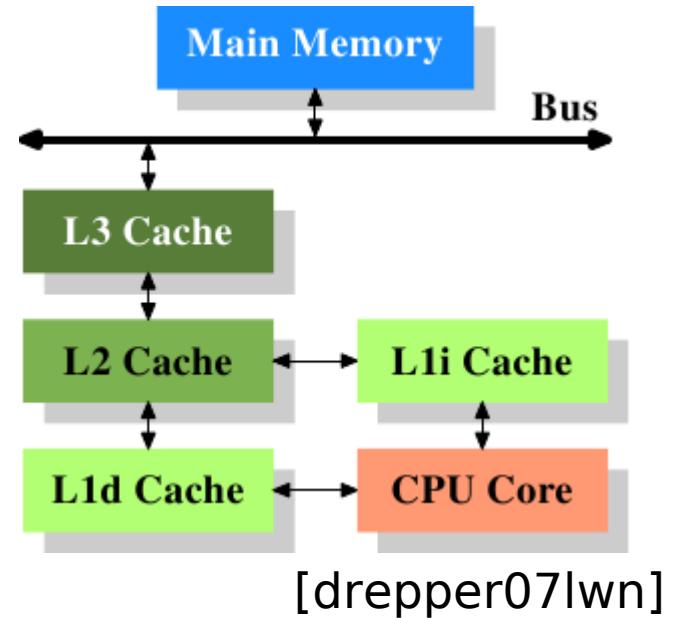
- memorijski zastoji mogu značajno smanjiti efektivni CPI
- performansa procesora raste brže od latencije memorije  
⇒ utjecaj promašaja na performansu **raste**
- svojstva priručne memorije ne mogu se zanemariti pri evaluiranju performanse sustava

# 6. Priručne memorije

1. Svojstva i organizacija dinamičkog RAM-a
2. Memorjska hijerarhija
3. Organizacija priručne memorije
4. Odabir parametara, performansa
5. **Izvedbeni detalji**

# Višerazinska priručna memorija

- Primarna PM (L1) spojena izravno na CPU (malena, brza)
- Sekundarna PM (L2) servisira L1 promašaje
  - veća, sporija, znatno brža od RAM-a
- RAM servisira L2 promašaje
- Sofisticirana računala imaju i PM L3
  - u višeprocesorskom sustavu, L3 obično dijele svi procesori
  - asocijativnost L3 mora biti veća od broja procesora!



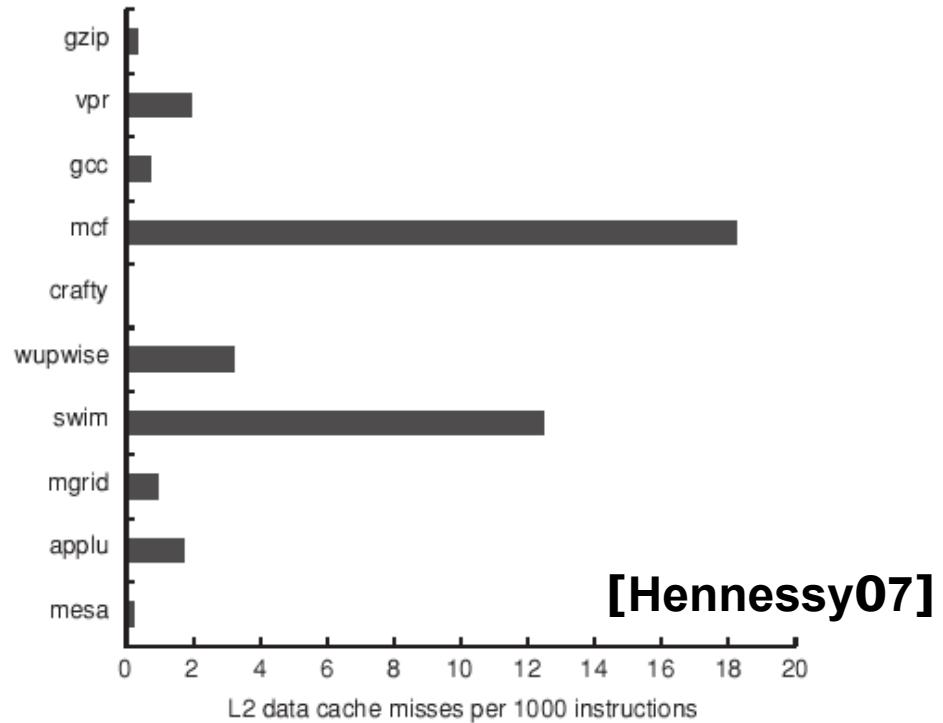
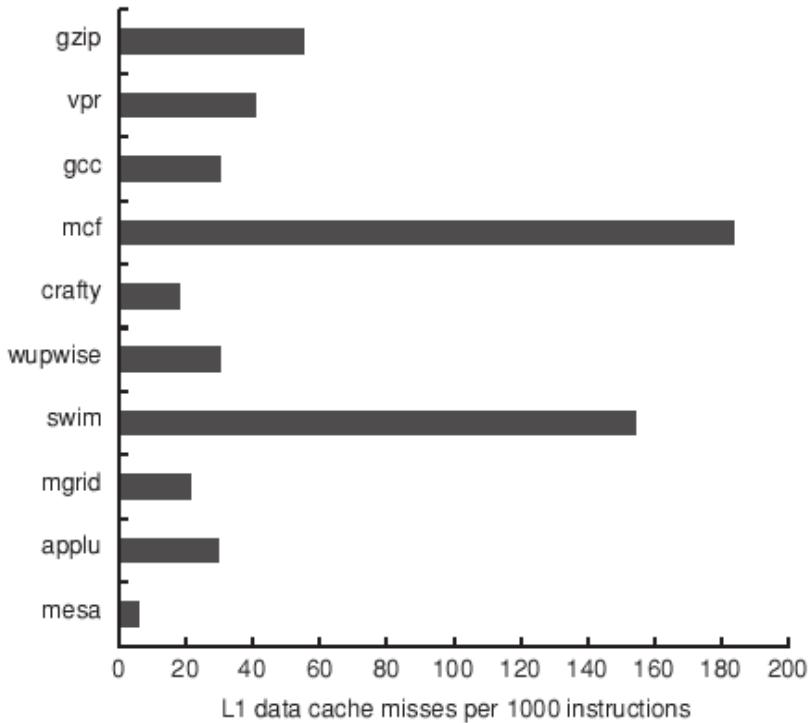
# Višerazinska priručna memorija (primjer)

- zadano:
  - $CPI_O = 1$ ,  $T = 250 \text{ ps}$
  - $v(\text{promašaj, L1}) = 2\%$
  - $t(\text{promašaj}) = 100 \text{ ns}$
- samo s PM L1:
  - $c(\text{promašaj}) = 100 \text{ ns} / 0.25 \text{ ns} = 400 \Delta T$
  - $CPI_{L1} = CPI_O + 400 \times 2\% = 9$
- svojstva L2
  - $t(\text{pogodak, L2}) = 5 \text{ ns} (20 \Delta T)$
  - $v(\text{promašaj, L2}) = 0.5\%$
- CPI s L1 i L2
  - $CPI_{L1+L2} = CPI_O + 2\% \times 20 \Delta T + 0.5\% \times 400 \Delta T$
  - $CPI_{L1+L2} = 1 + 0.4 + 2 = 3.4$

# Višerazinska priručna memorija (sažetak)

- PM L1: vrijeme pogotka prilagoditi taktu procesora
- PM L2: minimizirati učestalost promašaja
  - vrijeme pogotka manje interesantno
- Zaključci
  - PM L1 tipično manja od jedinstvene PM
  - linija L1 tipično manja od linije L2

# Promašaji PM podataka (P4, specCPU2000)

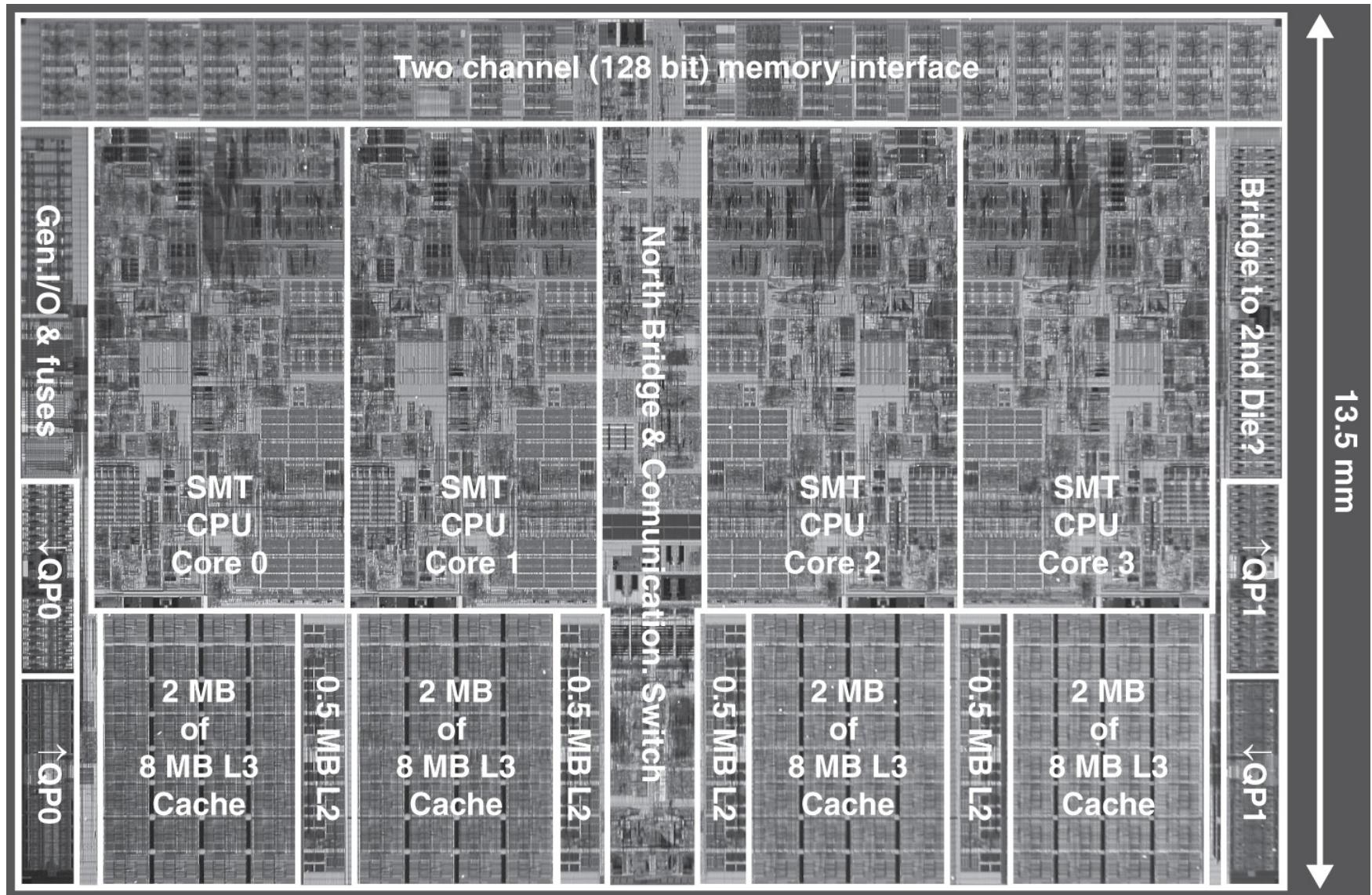


U istom eksperimentu, promašaji PMI zanemarivi (0.2-0.6 %)

# Priručne memorije, primjeri

- Intel Pentium 4 (Prescott, 2004):
  - I\$ 12 kμop ( $4 \times a$ ), D\$: 16 kB ( $8 \times a$ )
  - L2: 2 MB, ( $8 \times a$ )
- Ultra Sparc IV+ (2005)
  - I\$ 64 kB (64B,  $4 \times a$ ), D\$: 64 kB (32B,  $4 \times a$ )
  - L2: 2 MB (64B,  $4 \times a$ )
  - L3: 32 MB (64B,  $4 \times a$ )
- AMD Athlon 64 (2005)
  - I\$ 64 kB (64B,  $2 \times a$ ), D\$: 64 kB (64B,  $2 \times a$ )
  - L2: 1 MB ( $8 \times a$ )

# Intel Core i7: I\$ 32kB, D\$ 32kB, L2 512kB

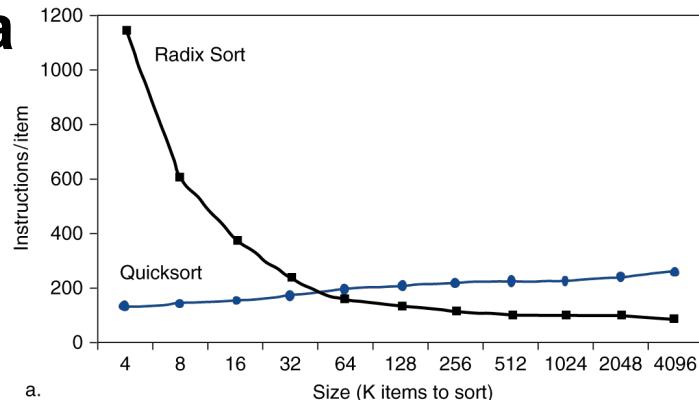


# PM u kontekstu dinamičkog raspoređivanja

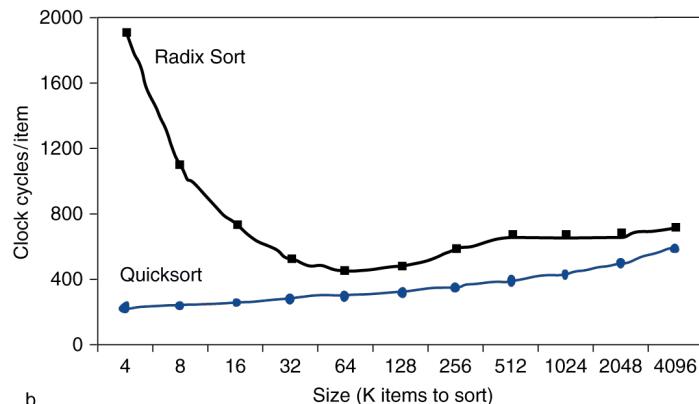
- Procesori s dinamičkim raspoređivanjem mogu raditi nešto korisno tijekom promašaja cachea!
  - instrukcije load/store čekaju PM u memorijskoj funkcijskoj jedinici
  - ovisne instrukcije čekaju u rezervacijskim redovima
  - neovisne instrukcije se nastavljaju izvoditi!
  - neki procesori omogućavaju više istovremenih pristupa PM!
- cijenu promašaja PM teško analizirati
  - učinak promašaja ovisi o strukturi programa
  - jednostavna procjena prosječnog trajanja pristupa memoriji (engl. AMAT) nije relevantna
  - do relevantnijih procjena može se doći simulacijom rada računalnog sustava

# PM u kontekstu zahtjevnih programa

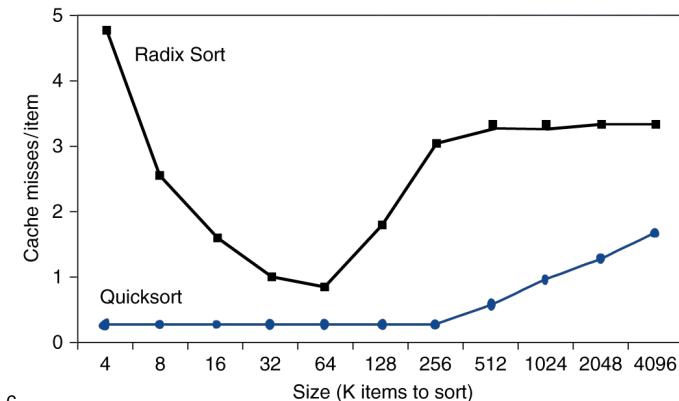
- promašaji ovise o redosljedu pristupa!
  - algoritamska prednost u O-notaciji može se istopiti uslijed suboptimalnog redosljeda pristupa podatcima
  - moderni prevoditelji mogu pomoći!
- prilagoditi strukture podataka liniji PM L1?
  - moderni procesori pružaju mogućnost dinamičkog prilagođavanja programa strukturi memorijskog sustava
  - x86: instrukcija cpuid!
- kakvu PM ima moje računalo?
  - Linux: hardinfo, x86info, cpuid
  - Windows: System Information Viewer



a.



b.



c.

# Priručne memorije, sažetak

- glavna ideja: ubrzati **najčešći** slučaj korištenjem **lokalnosti** pristupa
  - lokalnost pristupa: u zadanom vremenskom intervalu, programi koriste relativno mali dio ukupnog memoriskog prostora
  - brze memorije su malene, velike memorije su spore
  - memoriska hijerarhija nam često donosi najbolje od oba svijeta!
- koncept **cacheiranja** često se koristi u računarstvu:
  - datotečni sustav, preglednici weba, baze podataka
  - općeniti pristup: zapamtiti rezultat skupe operacije i koristiti ga pri naknadnim pozivima
- dimenzioniranje priručne memorije optimiranjem modela izvođenja
  - procjena prosječnog vremena pristupa
  - ovisi o najčešćim programima, tehnologiji, budžetu, ...

## 7. Organizacija moderne procesorske jezgre

- slična originalnom von Neumannovom konceptu
  - procesna jedinica (ALU)
  - upravljačka jedinica
  - memorija
  - periferija (ulaz, izlaz)
- **razlike:**
  - odmak od ograničenja **akumulatorske arhitekture**
    - skup registara opće i posebne namjene
  - **odvojene priručne memorije** (Harvard vs. Princeton)
    - Von Neumannova arhitektura: zajednička memorija za podatke i instrukcije (`load` inicira dva pristupa zajedničkoj memoriji!)
    - Harvardska arhitektura: **ispisna** instrukcijska memorija odvojena od podatkovne (kako upisati program u memoriju?)
    - vidjet ćemo kasnije zašto je Harvardska arhitektura **praktičnija**
  - **protočnost** i ostale metode iskorištavanja **instrukcijskog paralelizma**

## Sastavni dijelovi modernog procesora:

- jedna ili **više** procesnih jedinica (ALU) :
  - zbrajalo, posmačni sklop
  - množilo
  - djelilo
  - matematičke funkcije
- upravljačka jedinica: ožičena (ili i µprog)
- **skup registara:** uglavnom opće namjene
- **interne sabirnice**
- **priručne memorije**

## Put podataka (datapath):

- ALU, interne sabirnice, priručne memorije i registri
- ključan koncept **organizacije** (mikroarhitekture) procesora

## Čimbenici performanse procesora $t_{CPU}$ :

$$t_{CPU} = n_{instrukcija} \times n_{taktova/instrukcija} \times T_{takta}$$

Pristup oblikovanja puta podataka arhitekture RISC:

- CPI  $\approx 1$ ,  $T_{takta}$  prilagođen pristupu PM
- plitka, dobro popunjena protočnost
- maksimalno ubrzati slijed mikrooperacija koje koriste **najčešće instrukcije** (make the common case fast!)
  - pristup memoriji (load, store): 20%, 10% (svaka treća!)
  - aritmetika (add, shift, and, ...): 55% (svaka druga!)
  - grananje (jr, bcc, jal, ...): 15% (jedna od sedam!)

## Značajke RISC arhitekture:

- Jedine instrukcije za pristup memoriji su *load / store*
- Najsloženije adresiranje je **registarsko indirektno s pomakom** (ili indeksni adresni način)
- Pravilan (ortogonalan) instrukcijski skup  
(nema specijalnih registara, insktrukcije imaju sličnu složenost)
- Prioritet: maksimalno ubrzati najčešću operaciju:  
add r1, r2, r3

# MIPS: primjer procesora arhitekture RISC

## 1. Skup registara

- 32 32-bitna regista (uglavnom) opće namjene:  
\$0 - \$31
- registrima pridružena alternativna (mnemonička) imena

| Broj regista | Mnemoničko ime | Funkcija                     |
|--------------|----------------|------------------------------|
| \$0          | \$zero         | Konstantna vrijednost 0      |
| \$1          | \$at           | Rezerviran za asembler       |
| \$2 - \$3    | \$v0 - \$v1    | Rezultat funkcije / izraza   |
| \$4 - \$7    | \$a0 - \$a3    | Argumenti                    |
| \$8 - \$15   | \$t0 - \$t7    | Privremene vrijednosti       |
| \$16 - \$23  | \$s0 - \$s7    | Spremljene priv. vrijednosti |
| \$24 - \$25  | \$t8 - \$t9    | Privremene vrijednosti       |
| \$26 - \$27  | \$k0 - \$k1    | Rezervirano za jezgru OS     |
| \$28         | \$gp           | Globalno kazalo              |
| \$29         | \$sp           | Kazalo stoga                 |
| \$30         | \$fp           | Kazalo okvira                |
| \$31         | \$ra           | Povratna adresa              |

## 2. MIPS – skup instrukcija:

- Sve instrukcije fiksne duljine – 32 bita
- Tri osnovne kategorije instrukcija
  - (A) Memorejske
  - (B) Aritmetičko – logičke
  - (C) Instrukcije grananja
- Dodatno:
  - instrukcije za rad s brojevima s pomičnim zarezom (FP) – ne razmatramo

## 2. MIPS – skup instrukcija:

### (A) Memorejske instrukcije

npr.

```
lw $s1, 20($s2)          // load word; $s1 ← M[$s2+20]  
sw $s1, 20($s2)          // store word; M[$s2+20] ← $s1
```

Instrukcijski format – "I – tip":



- Memorejska adresa se formira pribrajanjem konstante (konst.) **predznačno proširene** na 32 bita sadržaju **baznog registra rs** (registarsko indirektno adresiranje s pomakom)
- Registar **rt** – **odredišni** registar u slučaju naredbe **lw**; **izvoršni** u slučaju naredbe **sw**

## 2. MIPS – skup instrukcija:

### (B.1) Aritmetičko – logičke instrukcije s registarskim operandima

npr.

|                      |  |
|----------------------|--|
| add \$s1, \$s2, \$s3 | // add; \$s1 $\leftarrow$ \$s2 + \$s3      |
| sub \$s1, \$s2, \$s3 | // subtract; \$s1 $\leftarrow$ \$s2 - \$s3 |
| and \$s1, \$s2, \$s3 | // and; \$s1 $\leftarrow$ \$s2 & \$s3      |
| or \$s1, \$s2, \$s3  | // or; \$s1 $\leftarrow$ \$s2   \$s3       |

Instrukcijski format – "R – tip":



Troadresne instrukcije; polja **rs** i **rt** određuju **izvorišne registre (operande)**, a polje **rd** **odredišni registar (rezultat)**

Polje **funct** određuje **funkciju ALU** (op kod je isti za sve A-L instrukcije);  
**shamt** (shift amount) **iznos posmaka** (koristi se samo u instrukcijama posmaka)

## 2. MIPS – skup instrukcija:

(B.2) Aritmetičko – logičke instrukcije s neposrednim operandom  
npr.

```
addi $s1, $s2, 1      // add; $s1 ← $s2 + 1
andi $s1, $s2, ff     // and; $s1 ← $s2 & ff
ori $s1, $s2, ff      // or; $s1 ← $s2 | ff
```

Instrukcijski format – "I – tip":



- Aritmetička ili logička operacija izvodi se između jednog operanda u registru **rs** i konstante (konst.) **predznačeno proširene na 32 bita**
- Rezultat se spremi u **odredišni registar rt**

## 2. MIPS – skup instrukcija:

### (C.1) Instrukcije grananja s absolutnom adresom

Bezuvjetno absolutno grananje – npr.

```
j 2500          // jump to target address; PC ← 2500  
jal 2500        // jump and link; $ra ← PC + 4, PC ← 2500
```

Instrukcijski format – "J – tip":



- izvođenje programa nastavlja se od adrese **adr \*4** (sve instrukcije imaju 32 bita).
- za jal povezni register **\$ra** je **implicitan**

## 2. MIPS – skup instrukcija:

### (C.2) Instrukcije grananja s relativnom adresom

Bezuvjetno registarsko grananje – npr.

```
jr $ra           // jump register; PC ← $ra
```

Uvjetno grananje – npr.

```
beq $t1, $t2, 25    // branch if equal;  
                    (ako je $t1 == $t2 onda PC ← (PC+4) + 25<<2)  
bne $t1, $t2, 25    // branch if not equal;  
                    (ako je $t1 != $t2 onda PC ← (PC+4) + 25<<2)
```

- **nema** statusnog registra: **uvjet** se ispituje nad **dvama registrima opće namjene**;
- grananje je **relativno** u odnosu na **PC+4**;

Koristi se "**I – tip**" instrukcije



## Primjeri

- Kako bi u memoriji izgledale instrukcije:

```
lw $t0, 1200($t1)          // op. kod 35
add $t0, $s2, $t0           // op. kod 0; f. kod za add = 32
sw $t0, 1200($t1)          // op. kod 43
bne $s0, $s1, -12           // op. kod 5
```

Odgovor:

- 100011 01001 01000 0000 0100 1011 0000 (hex. 8d2804b0)
- 000000 10010 01000 01000 00000 100000 (hex. 02484020)
- 101011 01001 01000 0000 0100 1011 0000 (hex. ad2804b0)
- 000101 10000 10001 1111 1111 1111 0100 (hex. 1611fff4)

Primjer: Koju instrukciju predstavlja sljedeći kod?

- **00af8020** (hex)

Odgovor:

- 00af8020 (hex) = 000000 00101 01111 10000 00000 100000 (bin)
- op. kod = 0 (troadresne aritmetičko-logičke instrukcije)
- rs = 5 ( registar \$5 = \$a1)
- rt = 15 (registar \$15 = \$t7)
- rd = 16 (registar \$16 = \$s0)
- shamt = 0 (iznos posmaka)
- funct = 32 (funkcija add)

=> riječ je o instrukciji **add \$s0, \$a1, \$t7**

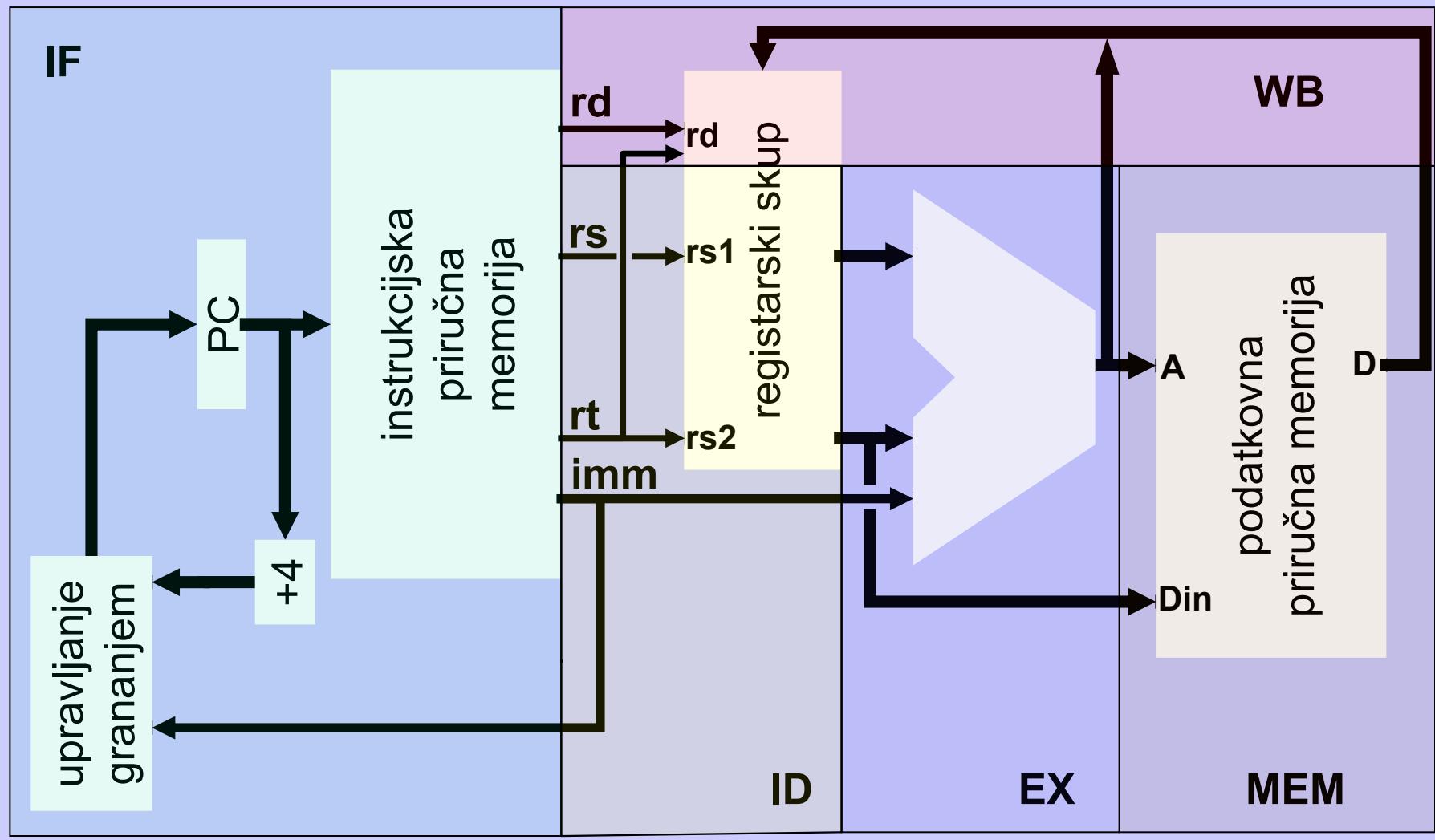
## Tijek izvođenja instrukcija iz tri najčešće grupe:

- prijavljanje instrukcije iz instrukcijske memorije (PC)
- prosljeđivanje kôdova registara (1 ili 2) registarskom skupu, čitanje registara
- u ovisnosti o vrsti instrukcije, ALU određuje:
  - rezultat aritmetičke operacije
  - efektivnu adresu
  - odredište relativnog grananja (MIPS: ne!)
- memorijeske instrukcije pristupaju memoriji
- upisivanje rezultata u odredišni registar

## Segmenti puta podataka arhitekture MIPS:

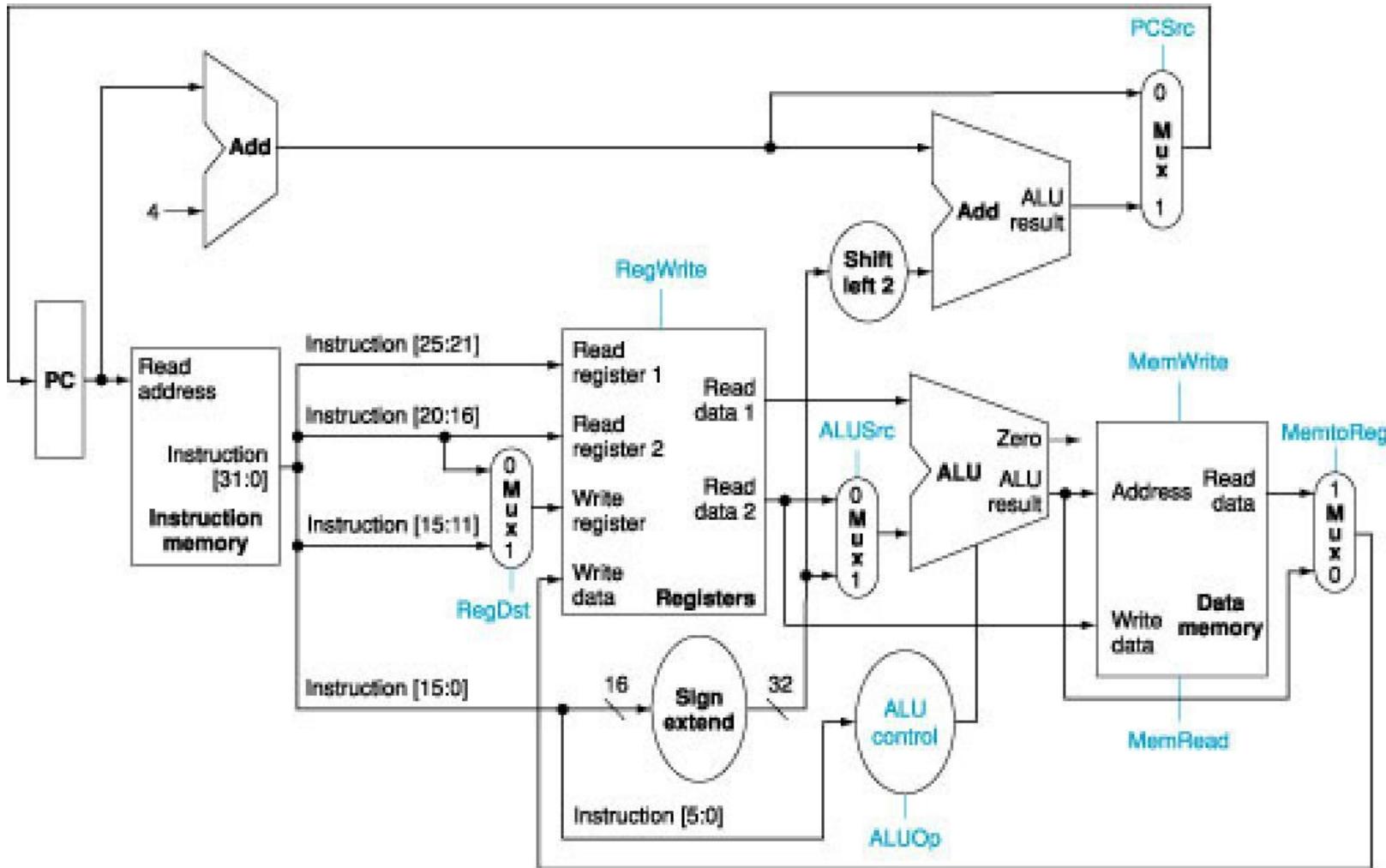
1. IF: pribavljanje 32-bitne instrukcije,  $PC=PC+4$
2. ID: dekodiranje operacijskog koda, pribavljanje registarskih operanada (0,1 ili 2)
3. EX: zbrajanje ili posmak  
(aritmetika, efektivna adresa)
4. MEM: pristup podatkovnoj memoriji  
(samo memorejske instrukcije)
5. WB: upis odredišnog registarskog operanda  
(aritmetika, load)

# Put podataka za računalo arhitekture MIPS - pojednostavljeno



# Detaljan prikaz puta podataka arhitekture MIPS

- za minimalni podskup skupa instrukcija:  
lw, sw, beq, add, sub, and, or, slt (I- i R- tip)

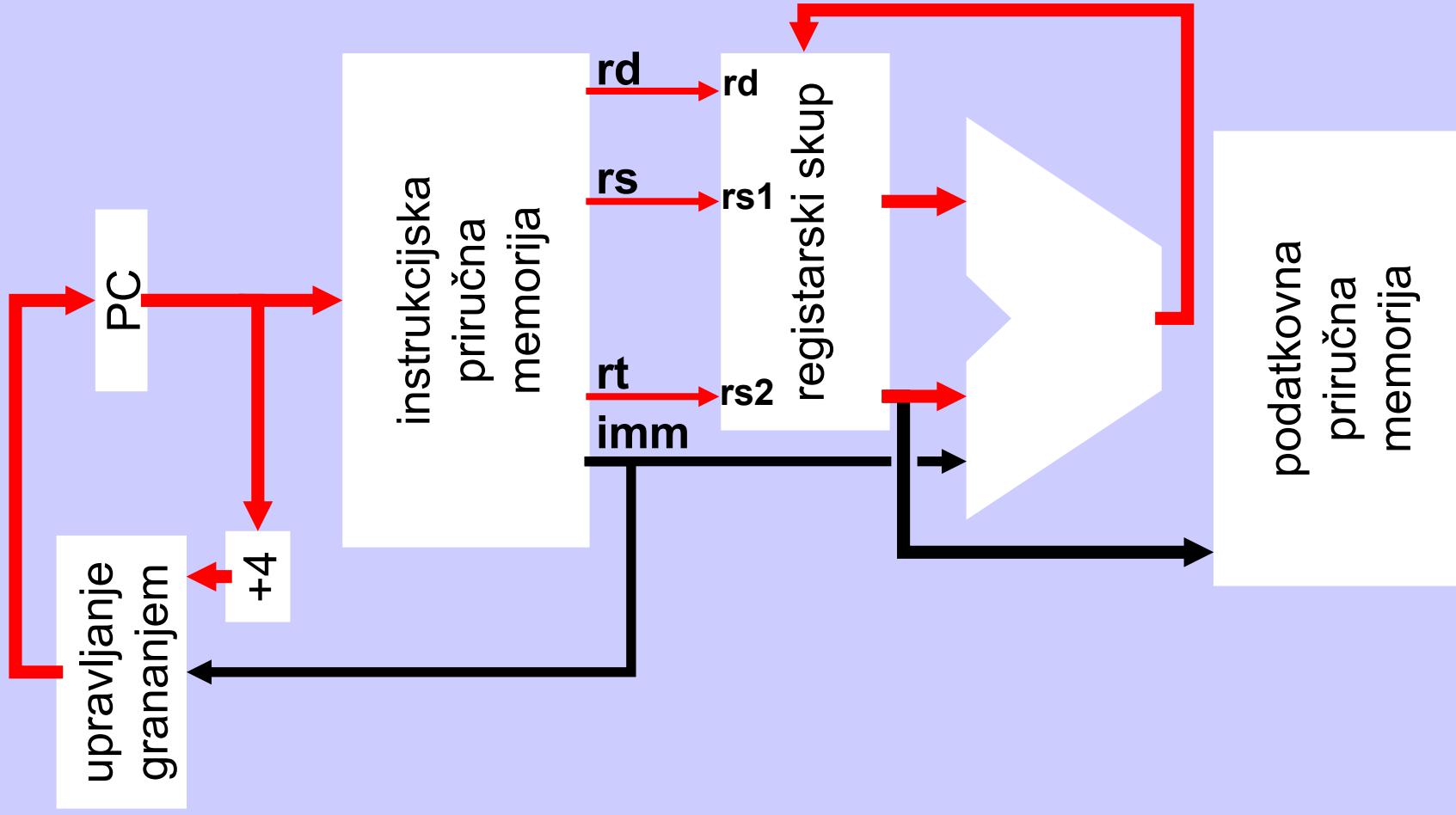


## Primjer 1: registarska aritmetička instrukcija

```
sub $rd, $rs,$rt    # $rd←$rs-$rt
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati op. kod, pribaviti \$rs i \$rt
- EX: oduzeti pribavljenе podatke
- MEM: ništa (nije memorijkska instrukcija)
- WB: upisati rezultat zbrajanja u \$rd

sub \$rd, \$rs, \$rt # \$rd←\$rs-\$rt



R-tip

000000

rs

rt

rd

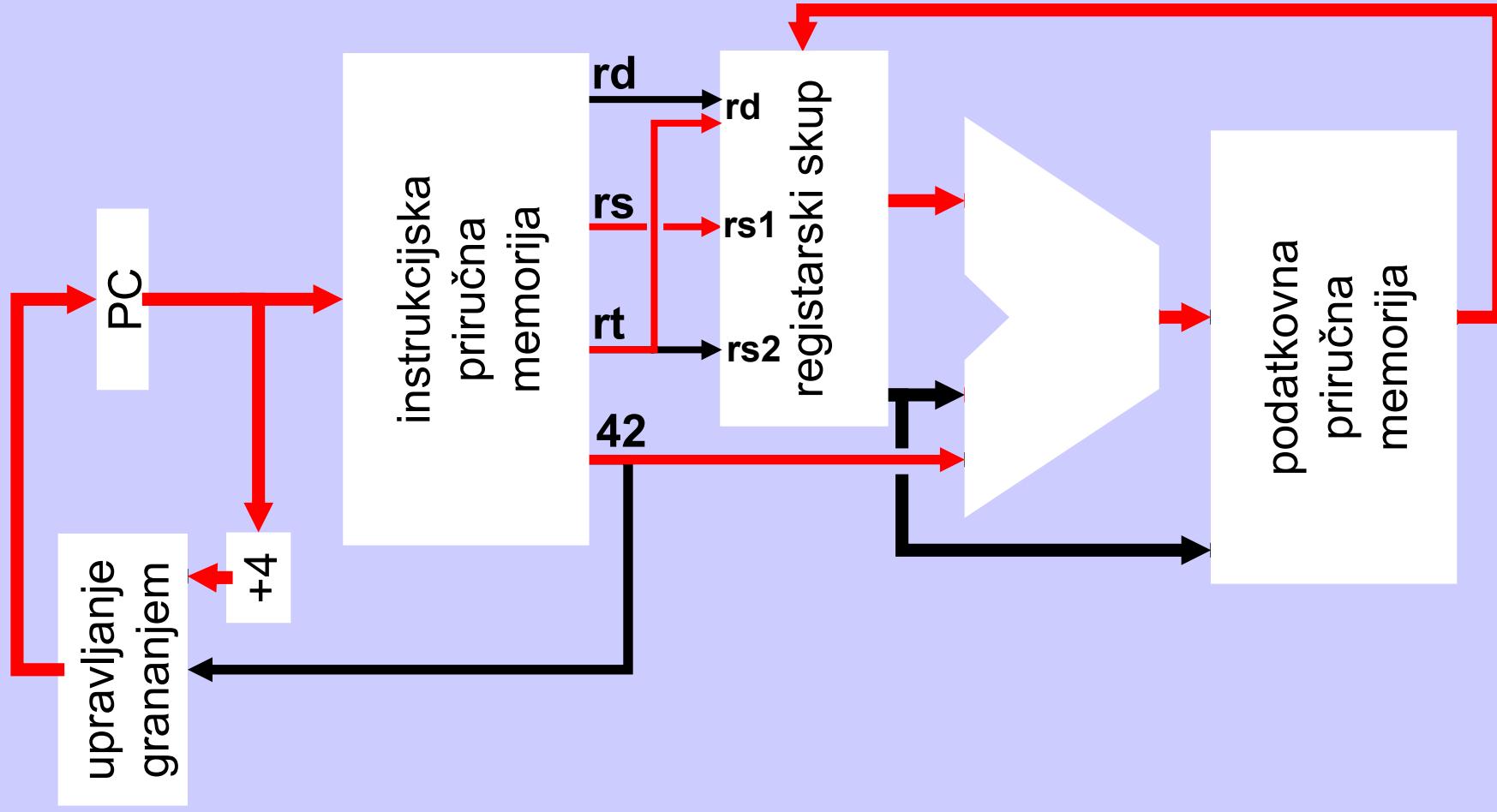
000000 100011

## Primjer 2: memorijska instrukcija

```
lw $rt, 42($rs)    # $rt←MEM($rs+42)
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati, pribaviti \$rs, proslijediti konstantu (42)
- EX: zbrojiti \$rs i 42
- MEM: učitati podatak (32 bit!) s adrese dobivene zbrajanjem
- WB: upisati učitani podatak u \$rt

`lw $rt, 42($rs) # $rt  $\leftarrow$  MEM[$rs+42]`



I-tip

100011

rs

rt

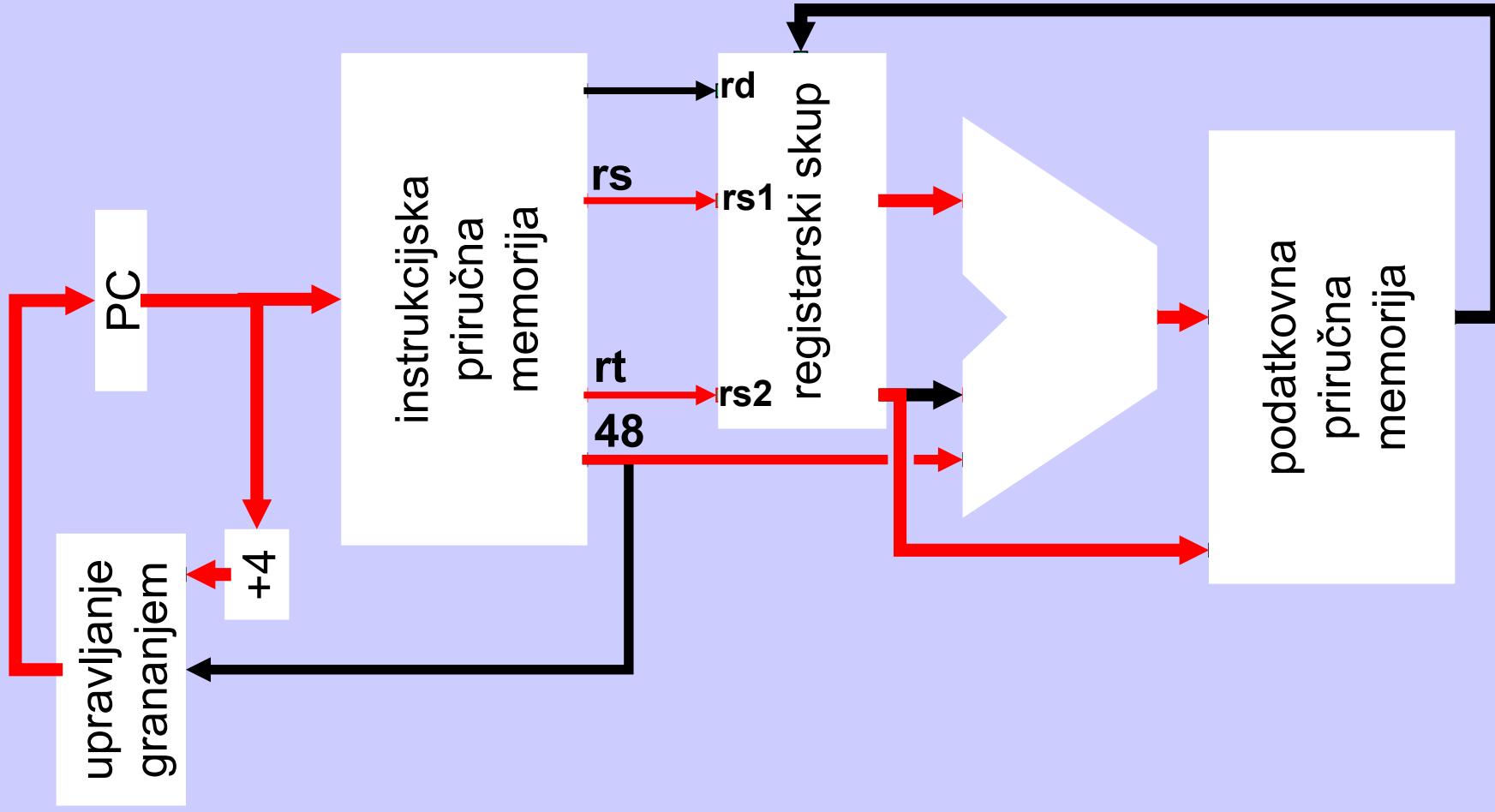
0000 0000 0100 0010

## Primjer 3: memorijska instrukcija

```
sw $rt,48($rs)    # MEM($rs+48)←$rt
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati, pribaviti \$rs, proslijediti 48
- EX: zbrojiti \$rs i 48
- MEM: upisati podatak (32 bit!) iz registra \$rt na adresu dobivenu zbrajanjem
- WB: ništa

```
sw $rt, 48($rs) # MEM[$rs+48]←$rt
```



I-tip

101011

rs

rt

0000 0000 0100 1000

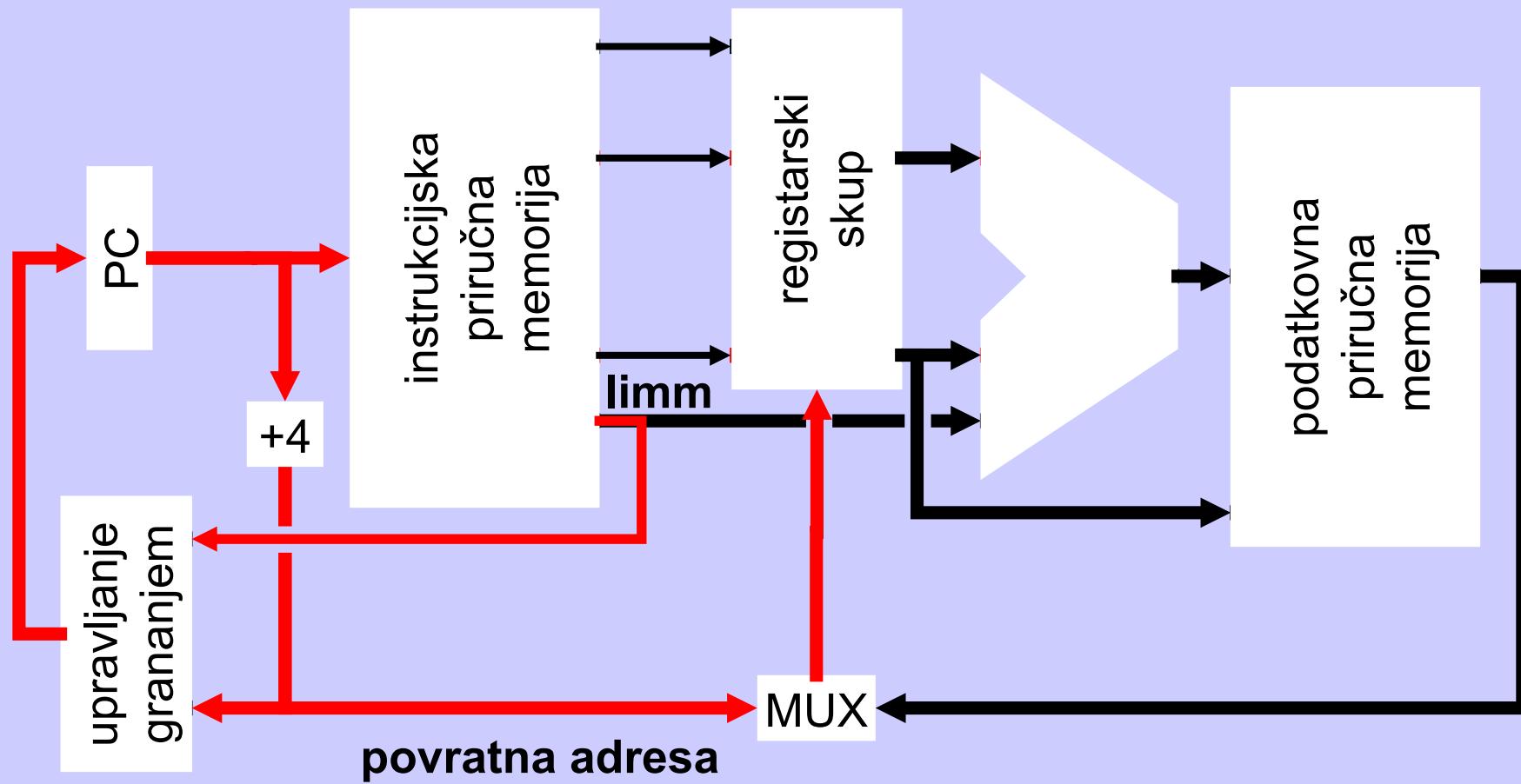
## Primjer 4: instrukcija grananja:

```
jal $10000 # $pc←$10000; $r31←$pc+4
```

- IF: pribaviti instrukciju,  $PC += 4$
- ID: dekodirati, proslijediti  $\$10000$  izravno u PC
- EX: ništa
- MEM: ništa
- WB: upisati povratnu adresu u  $\$r31$

**Latencija** ovakvog grananja će otežati protočnu izvedbu!

jal \$10000 # \$pc←\$10000; \$r31←\$pc+4\*



J-tip\*\* 000011 00 0000 0000 0100 0000 0000 0000

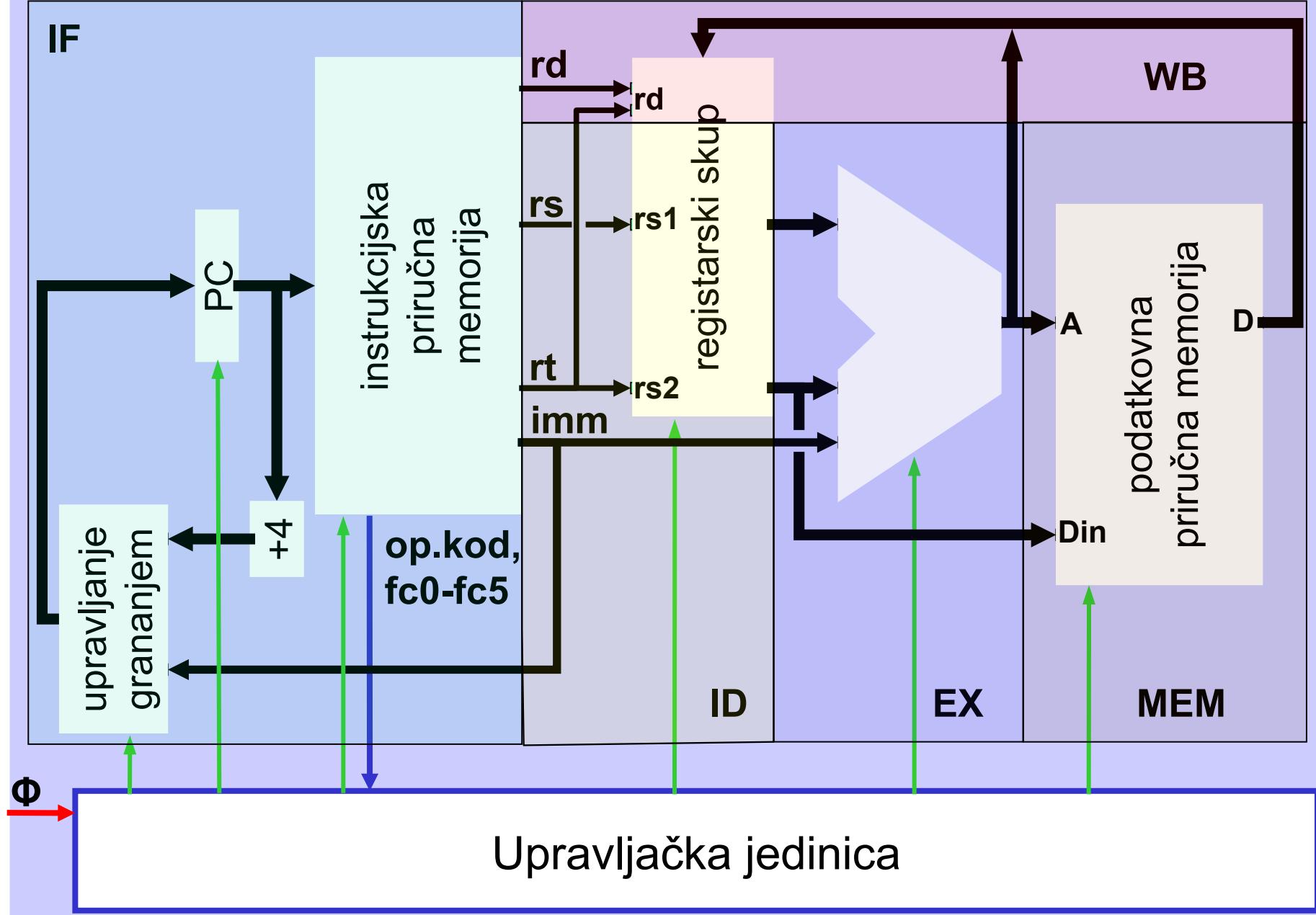
\* u zakašnjeloj izvedbi PC pokazuje na priključak za kašnjenje

\*\* limm se množi s 4!

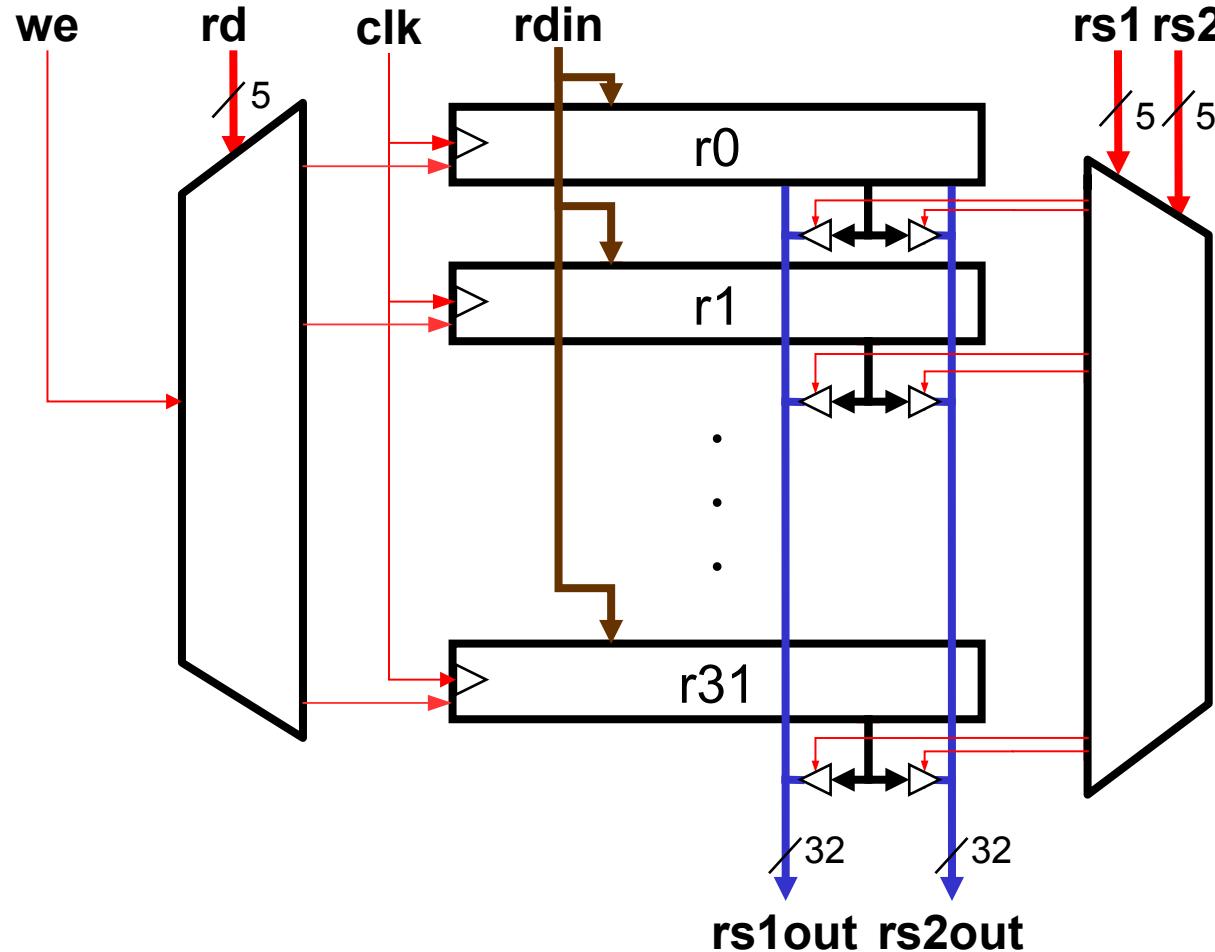
# Svojstva puta podataka arhitekture MIPS:

- omogućava sve mikrooperacije potrebne za izvedbu najvažnijih instrukcija ISA-e
  - ideja: maksimalno ubrzati jednostavne instrukcije
  - pet segmenata, samo instrukcije tipa load koriste sve segmente
- potrebno sklopolje:
  - dekoderi i multiplekseri
  - interne sabirnice
  - registrski skup
  - paralelno zbrajalo (o tome ste čuli na Digitalnoj elektronici)
  - priručne memorije (prethodno predavanje!)
- pravilan protok podataka osigurava jednostavno i efikasno upravljanje!

# Arhitektura CPU: put podataka+upravljanje



# Izvedba skupa registara opće namjene

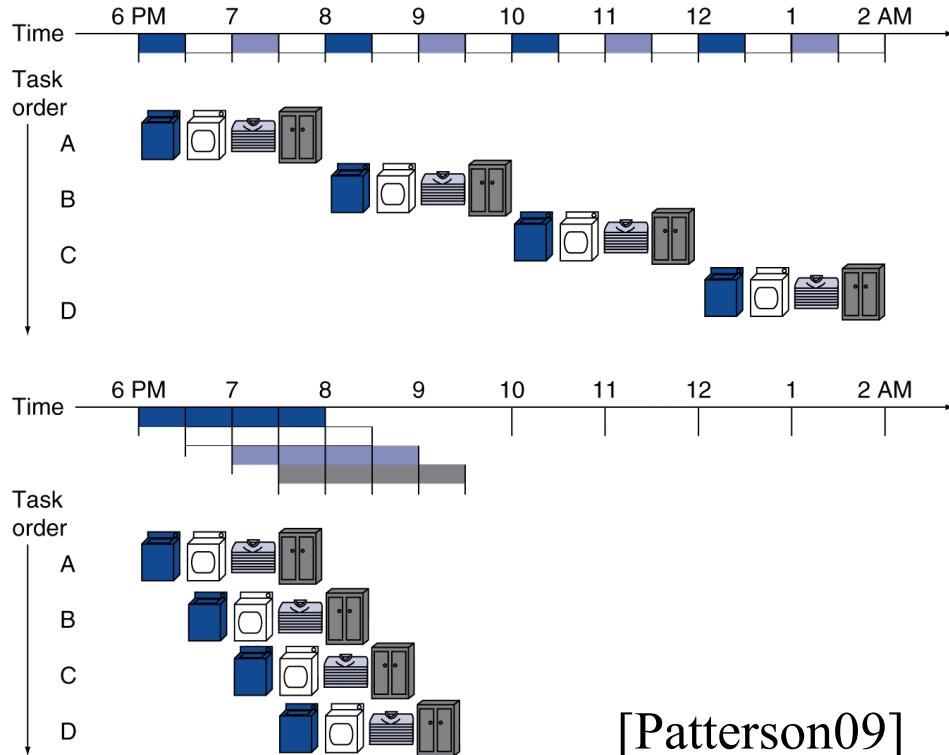


# Pogled na performansu

- trajanje instrukcije određuje najdulje kašnjenje
  - kritični put: instrukcija load
  - instrukcijska memorija → registri → ALU → podatkovna memorija → registri
- možemo li popraviti performansu bez skraćenja trajanja instrukcije?
  - da, ako uspijemo izvoditi **više instrukcija istovremeno**
  - potencijal **instrukcijskog paralelizma**: performansa  $\uparrow \times 10$ 
    - desetljeće ispred verzije bez instrukcijskog paralelizma, pod pretpostavkom godišnjeg porasta performanse od 25% ( $1.25^{10} \approx 10$ )!
  - najjednostavnija tehnika iskorištavanja ILP-a: protočnost

# Protočna organizacija na primjeru praonice rublja

- sastavni dijelovi **posla**: pranje, sušenje, glačanje, slaganje
- osnovna organizacija (gore): praonica se otpušta tek kad je **posao** gotov
- protočna organizacija (dolje): više **poslova** napreduje usporedno



- u slučaju četiri posla:  
 $\text{ubrzanje} = 8/3.5 = \mathbf{2.3}$
- kontinuirani rad (**n**):  
 $\text{ubrzanje} = 2n/(0.5n+1.5) \approx \mathbf{4}$   
= broj segmenata!
- ideja posuđena iz manufaktura i proizvodnih linija [Chaplin 1936]

[Patterson09]

# Protočna organizacija arhitekture MIPS

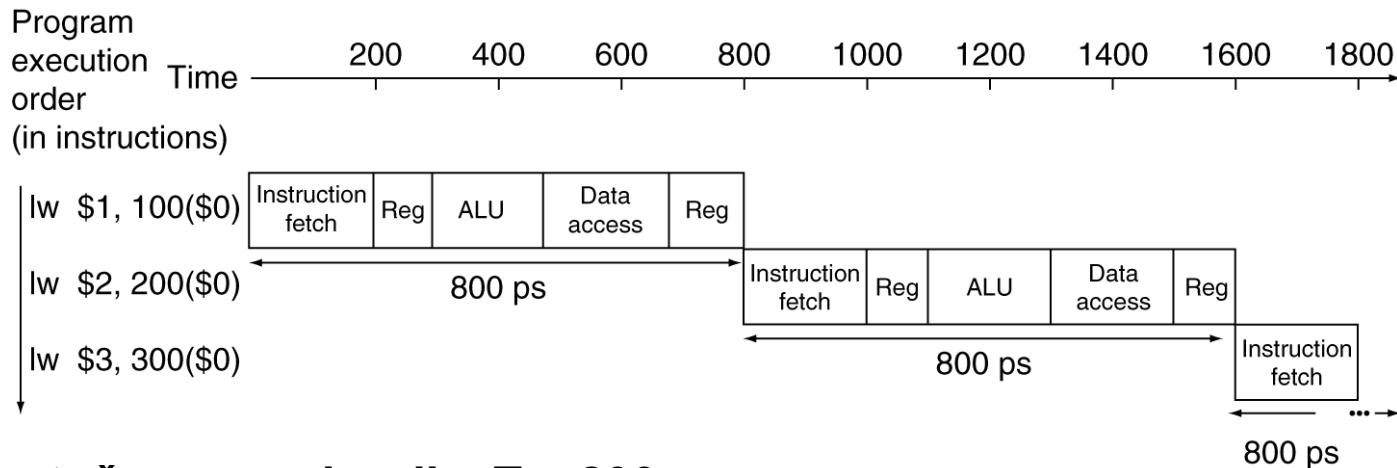
- pretpostavimo da pristup registrima traje 100ps, a ostale operacije 200ps
- tada dobivamo sljedeće latencije tipičnih instrukcija:

| instrukcija | instrukcijska<br>memorija | čitanje<br>registara | zbrajanje | memorija<br>podataka | upis<br>registra | latencija<br>(zadržavanje) |
|-------------|---------------------------|----------------------|-----------|----------------------|------------------|----------------------------|
| lw          | 200ps                     | 100 ps               | 200ps     | 200ps                | 100 ps           | 800ps                      |
| sw          | 200ps                     | 100 ps               | 200ps     | 200ps                |                  | 700ps                      |
| add         | 200ps                     | 100 ps               | 200ps     |                      | 100 ps           | 600ps                      |
| beq         | 200ps                     | 100 ps               | 200ps     |                      | 0 ps             | 500ps                      |

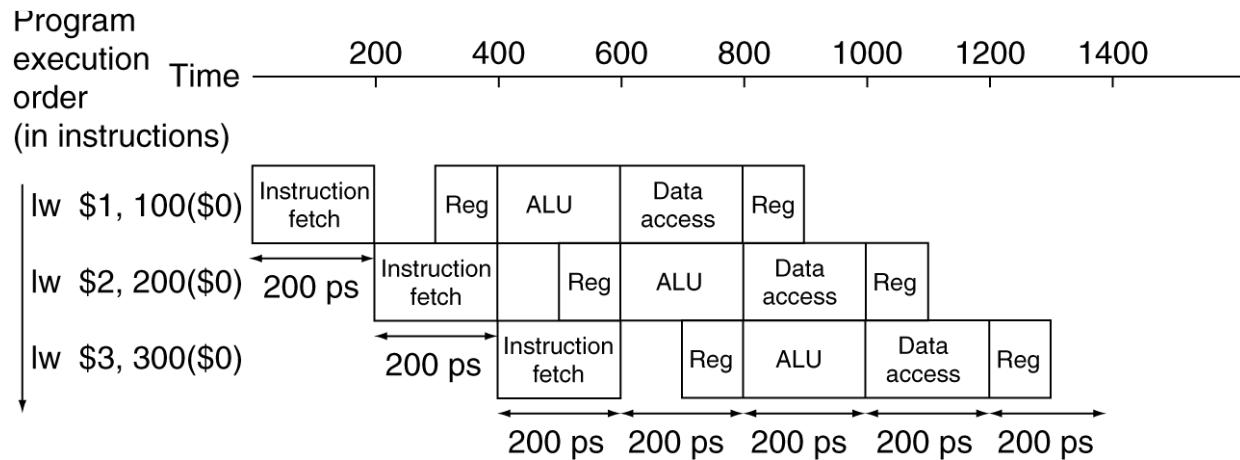
[Patterson09]

- što bi se dogodilo kad bismo na procesoru primijenili recept iz praonice?

## osnovna organizacija: $T_c = 800\text{ps}$



## protočna organizacija: $T_c = 200\text{ps}$



[Patterson09]

U prvoj aproksimaciji, za veći broj instrukcija dobili bismo ubrzanje  $\times 4$  !

# MIPS ISA je prilagođena protočnom konceptu

- sve instrukcije imaju 32 bita
  - dohvaćaju se u jednom ciklusu (poravnate s granicom riječi!)
- pravilni instrukcijski format
  - dekodiranje i čitanje registara u jednom ciklusu!
- pristup memoriji instrukcijama load i store
  - treći segment računa adresu, četvrти adresira memoriju
- memorijski operandi poravnati
  - pristup operandu u jednom ciklusu (ako je u priručnoj memoriji)

# Efekt protočne organizacije

- ako obrada u svakom od n segmenata jednako traje, ubrzanje = n
  - to je **idealni slučaj** kojem se ne možemo nadati
    - prethodna stranica 5 segmenata  $\Rightarrow$  ubrzanje najviše četverostruko
- inače, ubrzanje je manje
  - u našem **školskom slučaju**, ubrzanje ovisi o razdiobi instrukcija
- važno: ubrzanje postižemo uslijed veće propusnosti
  - trajanje izvođenja instrukcija se ne mijenja
  - ako želimo biti točni, izvođenje instrukcija se čak malo povećava!
- u **realnom slučaju** stvari će se dodatno zakomplicirati
  - **međuvisnost instrukcija** glavna kočnica za iskorištavanje instrukcijskog paralelizma
  - poremećaje koji sprječavaju glatki protok instrukcija jednim imenom nazivamo **hazardima**

**Hazard** – situacija koja izaziva poremećaje i kašnjenje u “glatkom” protoku zadataka kroz protočnu strukturu

Hazardi sprečavaju da sljedeća instrukcija u nizu bude izvedena u za nju predviđenom periodu signala takta.

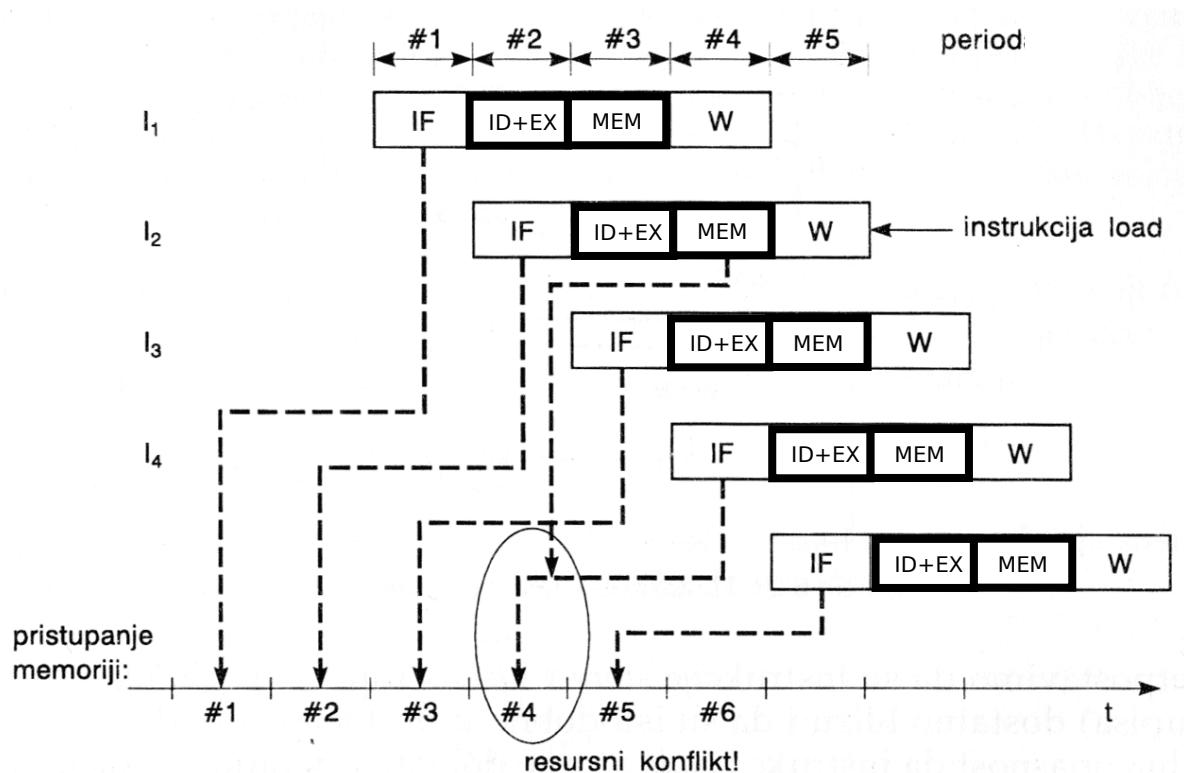
Tri razreda hazarda:

- strukturni hazard
- podatkovni hazard
- upravljački hazard

# Struktturni hazard (resursni konflikt)

Nastupa kad se instrukcija ne može izvesti zbog sukoba oko sredstava (resursa)

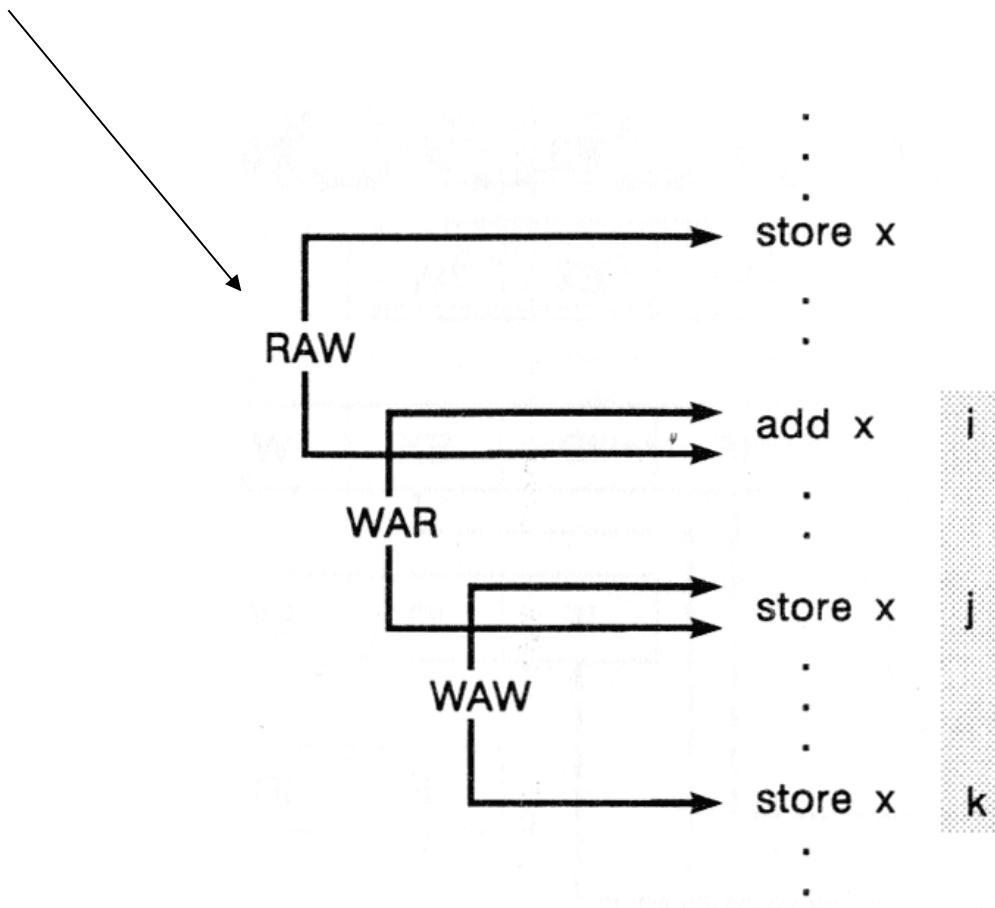
Primjer: konflikt kod pristupa zajedničkoj memoriji kod arhitekture Berkeley RISC (4 protočna segmenta):



# Podatkovni hazard

- Podatkovni hazard nastupa zbog *međuovisnosti podataka*
- Nastaje kad dvije ili više instrukcija pristupaju istom podatku ili modificiraju isti podatak
- Tri vrste podatkovnih hazarda:
  1. RAW – read after write /čitanje poslije upisa/
  2. WAR – write after read /pisanje poslije čitanja/
  3. WAW – write after write /pisanje poslije pisanja/
- protočne arhitekture upisuju rezultat u odredišni registar pri kraju instrukcijskog ciklusa pa nemaju hazarde WAW i WAR
- WAR i WAW nastaju isključivo u superskalarnim arhitekturama s dinamičkim raspoređivanjem i izvođenjem izvan redosljeda

RAW – postoji opasnost da instrukcija *add x* dohvati operand s lokacije *x* prije negoli instrukcija *store x* upiše novu vrijednost na lokaciji *x*



ekvivalentni registarski primjer:

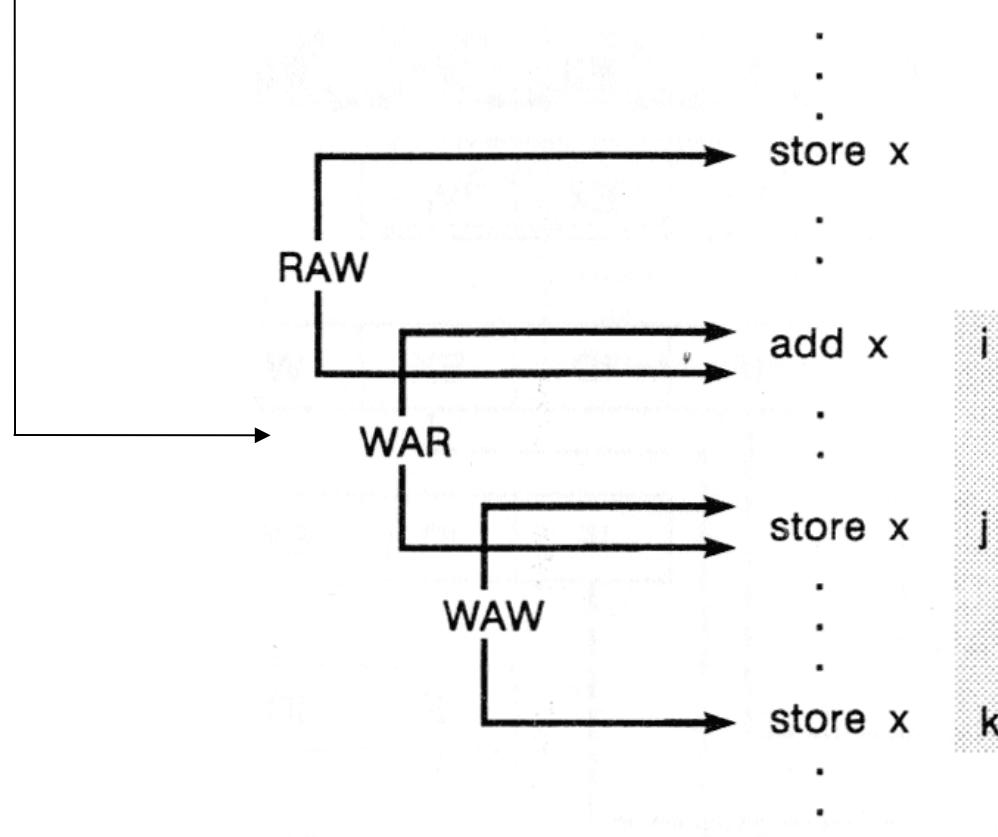
add \$r1, \$r2,\$r3

add \$r4, \$r1,\$r5#raw \$r1

add \$r5, \$r6,\$r7#war \$r5

lw \$r5, \$r0,40 #waw \$r5

WAR – instrukcija  $j$  ( $store\ x$ ) koja logički slijedi instrukciji  $i$  mijenja podatak na lokaciji  $x$  koju čita instrukcija  $i$   
(ovaj problem se ne javlja kod jednostavnih protičnih arhitektura)



registarski primjer:

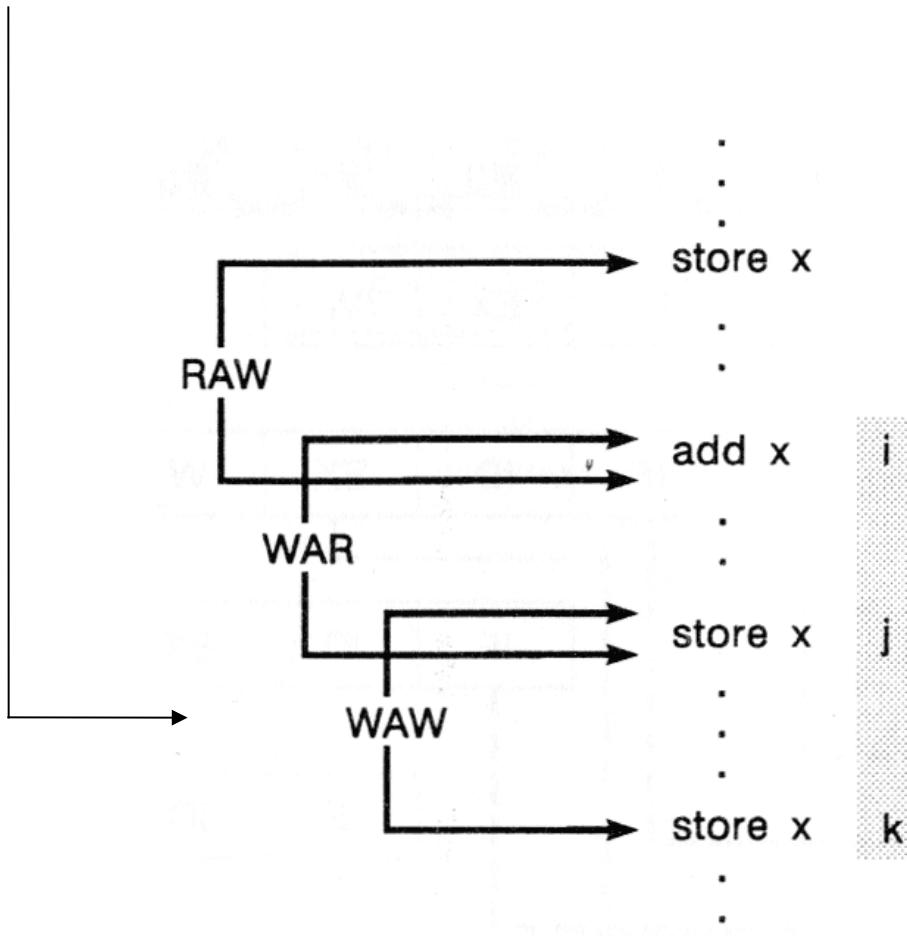
add \$r1, \$r2,\$r3

add \$r4, \$r1,\$r5#raw \$r1

add \$r5, \$r6,\$r7#war \$r5

lw \$r5, \$r0,40 #waw \$r5

WAW – obje instrukcije  $j$  i  $k$  žele upisati podatak na memorijskoj lokaciji  $x$   
(problem nastaje ako se instrukcija  $j$  izvede poslije instrukcije  $k$ ,  
ovaj problem se ne javlja kod jednostavnih protičnih arhitektura)



registarski primjer:

add \$r1,\$r2,\$r3

add \$r4, \$r1,\$r5 #raw \$r1

add \$r5, \$r6,\$r7 #war \$r5

lw \$r5, \$r0,40 #waw \$r5

Hazard vrste RAW je u protočnim arhitekturama najviše izražen tijekom izvođenja instrukcije *load*

load r1, A



load r2, B



add r3, r1, r2



# Saniranje hazarda RAW nakon instrukcije load:

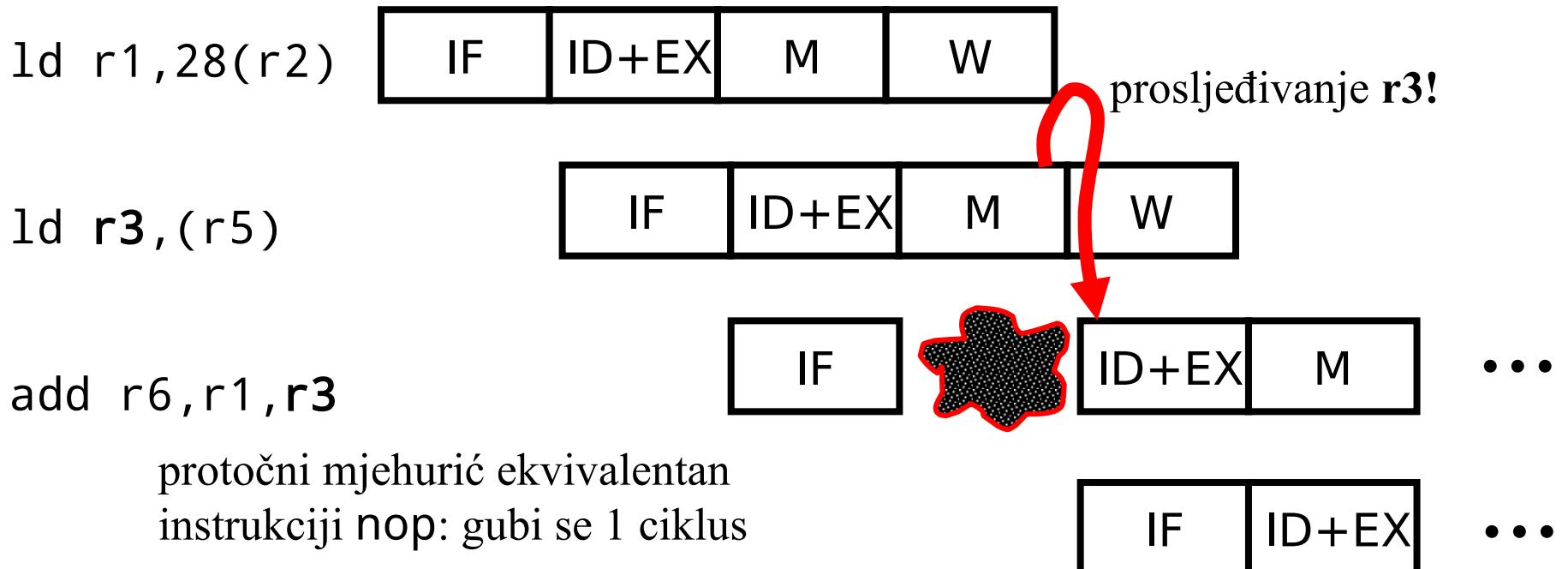
- usporavanjem protočnosti (**protočni mjehurići**, vidi dolje)
- **prosljedivanjem** npr,  $\text{MEM}[i] \rightarrow \text{EX}[i+1]$  (ipak jedan mjehurić)
- specifičnom definicijom instrukcije load (**zakašnjelo čitanje**)

*load       $r5 = \text{mem}(r1 + r2)$*   
*load       $r6 = \text{mem}(r3 + \text{offset})$*   
*add         $r7 = r5, r6$*   
*store      $\text{mem}(r1 + r2) = r7$*  .



Kod jednostavnih instrukcija, najčešće možemo proći samo s jednim mjehurićem

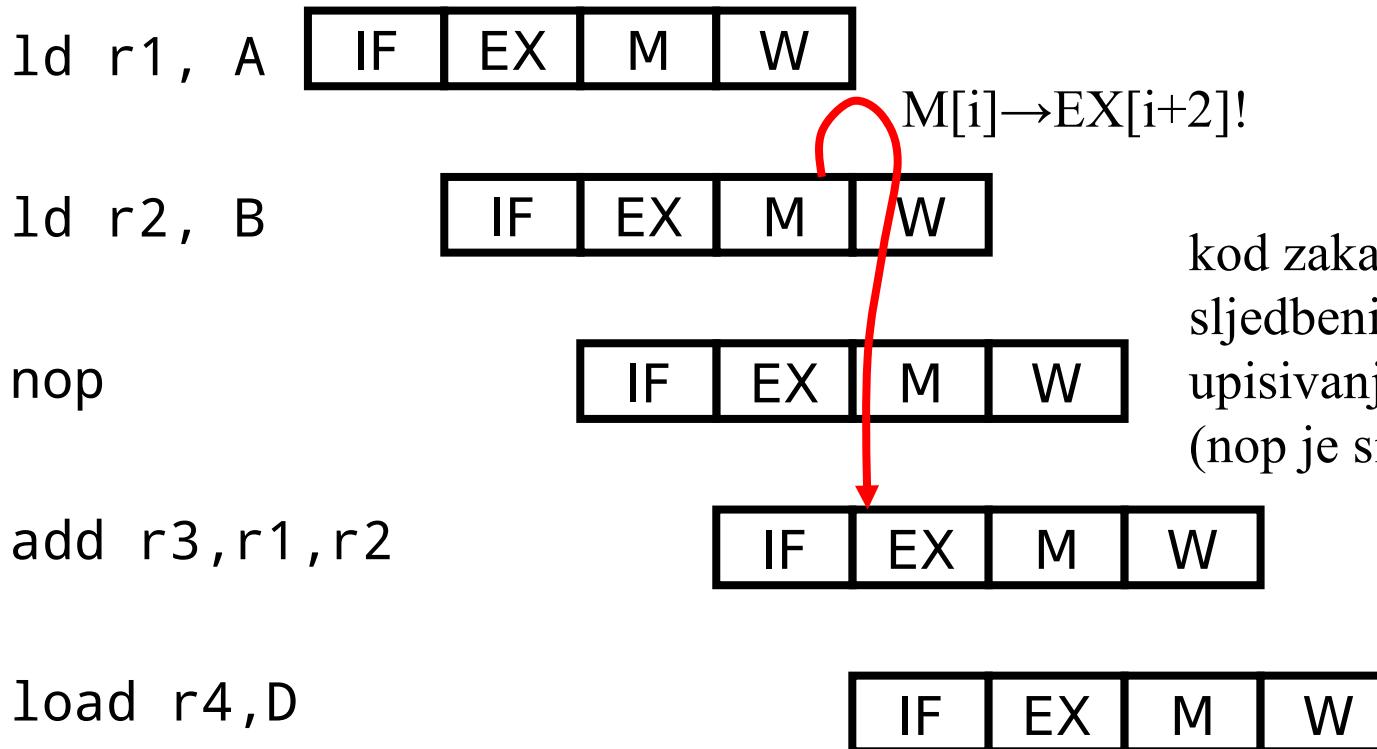
- **prosljeđivanje:** prospajanje (vodovi + MUX) rezultata prethodne operacije natrag prema ulazu procesne jedinice
- primjer za arhitekturu Berkeley RISC  $M[i] \rightarrow ID+EX[i+1]$ :



Ideja: potpuno otkloniti potrebu za mjeđučimama **zakašnjelom definicijom** problematičnih instrukcija (**load, jump**)

U takvoj arhitekturi prevodioc ima priliku **iza** zakašnjelih instrukcija staviti korisnu instrukciju, koja ne ovisi o rezultatu zakašnjele instrukcije

Logički, dodana instrukcija se izvršava **istovremeno** sa zakašnjelom instrukcijom.



kod zakašnjelih instrukcija,  
sljedbenica se izvodi prije  
upisivanja rezultata  
(nop je siguran izbor)!

Mjesto instrukcije u slijedu instrukcija neposredno nakon zakašnjele instrukcije (*load*) naziva se “priključak za kašnjenje” (engl. delay slot)

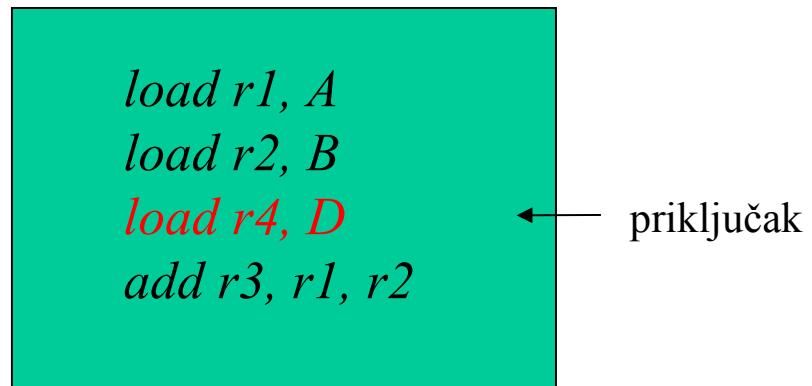
Ako prevodioc u priključak za kašnjenje ne uspije ubaciti korisnu instrukciju, ubacuje se instrukcija nop

Instrukcija iz priključka za kašnjenje ne vidi rezultate izvođenja prethodne instrukcije: logički, dvije instrukcije se izvršavaju **istovremeno**

Primjer (zakašnjelo čitanje):

$$C := A + B; E := D$$

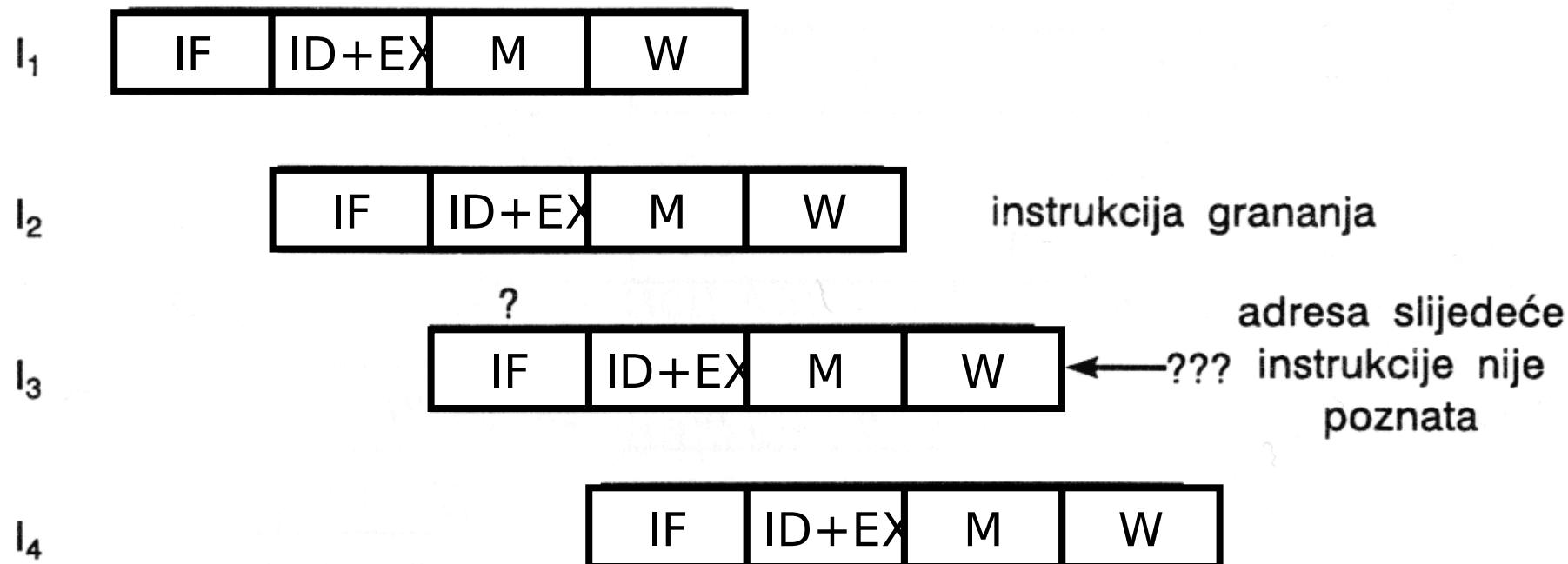
*load r1, A*  
*load r2, B*  
*add r3, r1, r2*      ← hazard RAW  
*load r4, D*



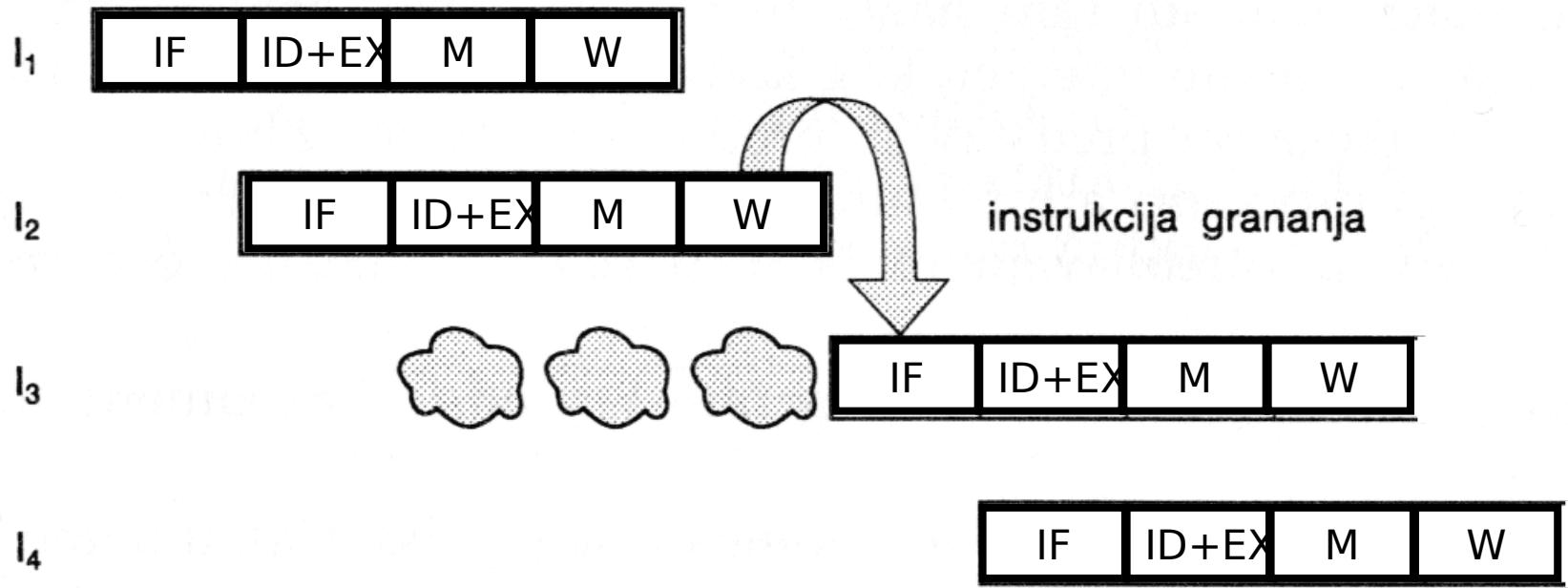
# Upravljački hazard

Nastupa kad adresu sljedeće instrukcije nije moguće izračunati prije njenog dohvata

Primjer:



Ako odredišnu adresu u PC upisujemo u segmentu W,  
potrebno je umetnuti tri (!!) protočna mjehurića:



Nepovoljno utječe na performansu procesora!

Smanjenje kašnjenja može se postići tako da se računanje i upis ciljne adrese grananja obavi u segmentu **ID** (umjesto u W ili EX)

Uz protočni mjehurić, koristi se **prosljeđivanje**  $ID[i] \rightarrow IF[i+1]$ : relativno odredište računa se u zasebnom zbrajalu, u okviru sklopa za upravljanje grananjem  
(glavno zbrajalo u to vrijeme računa rezultat prethodne operacije!)

Spekulativna operacija zbrajanja programskog brojila PC i konstante imm započinje dok instrukcija još nije dekodirana!

Rezultat dekodiranja instrukcije utječe na izlazni multiplekser koji konačno odabire između  $PC+4$  i  $PC+4+imm$

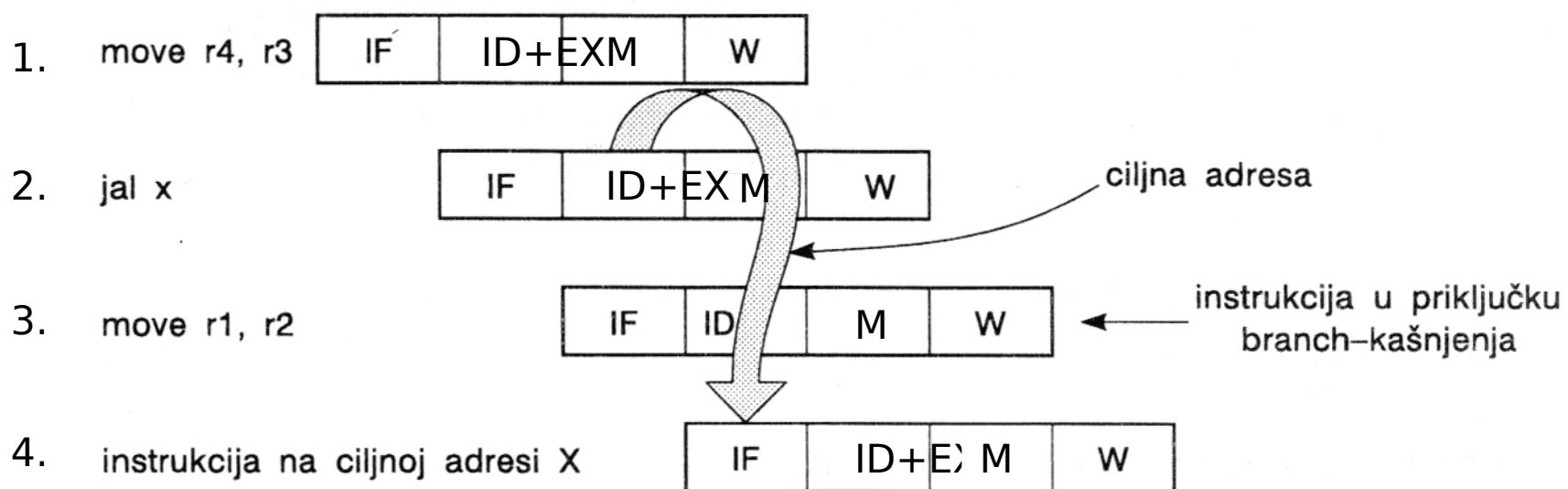
Kašnjenje samo s jednim mjehurićem!

Mjehurići se mogu potpuno zaobići **zakašnjelim grananjem**

- u tom slučaju koristimo prosljeđivanje  $ID[i] \rightarrow IF[i+2]$

## Zakašnjelo grananje:

- nema protočnih mjehurića
- proslijedivanje ID[2] → IF[4]



# Primjer za računalo MIPS: uvjetno relativno grananje

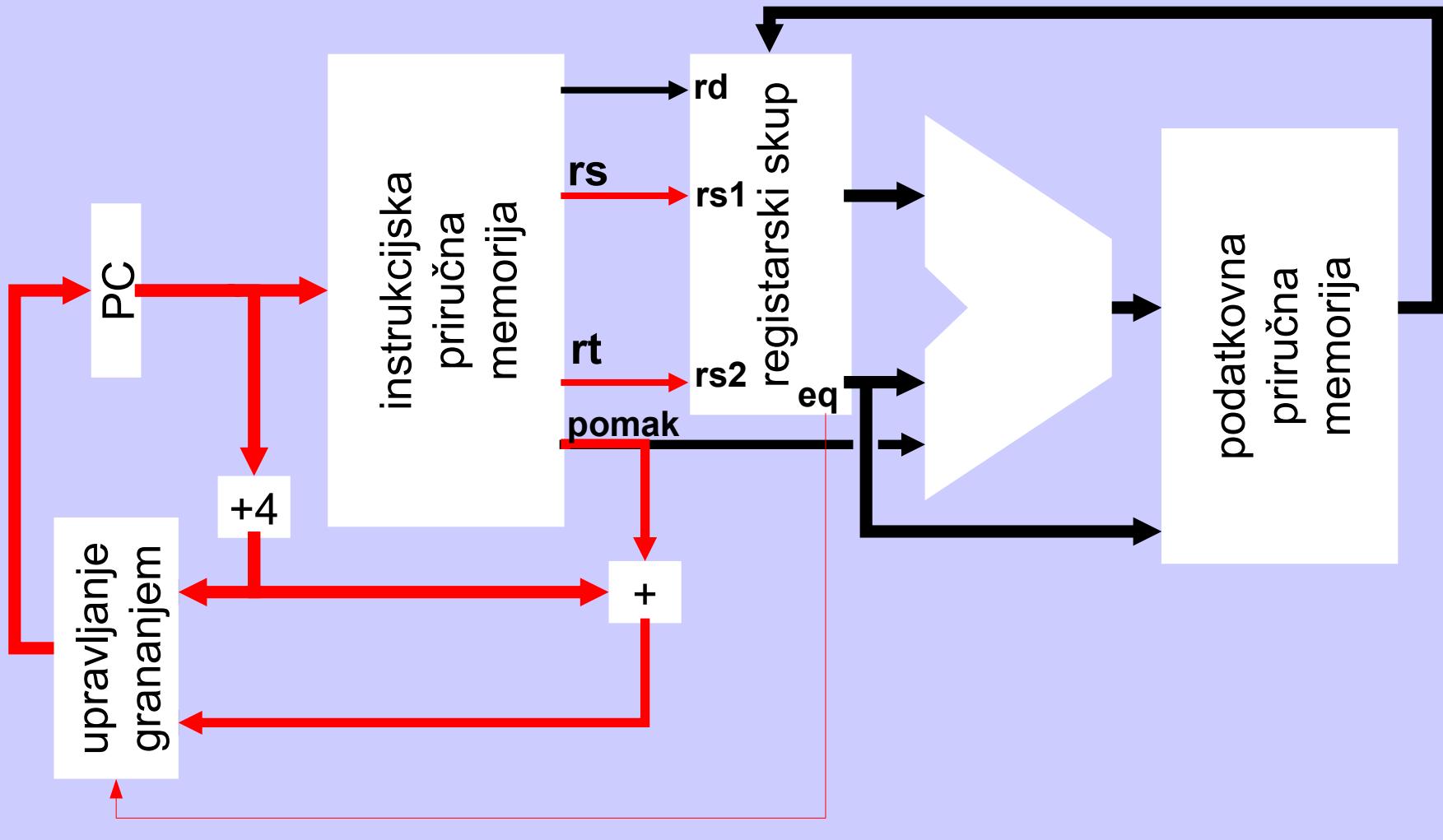
bne \$r1,\$r2,pomak #  $PC \leftarrow PC + 4 + \text{pomak}$ , ako ( $\$r1 \neq \$r2$ )

- IF: pribaviti instrukciju, odrediti  $PC + 4$
- ID:
  - dekodirati operacijski kod,
  - pribaviti  $\$r1, \$r2$ ,
  - usporediti  $\$r1, \$r2$ ,
  - zbrojiti  $PC + 4$  i pomak
  - upisati novi PC ( $PC + 4$  ili  $PC + 4 + \text{pomak}$ )
- EX, MEM, WB: ništa

## Komentari:

- instrukcija neposredno nakon instrukcije grananja se također izvodi
- ako želimo kasniti samo jedan takt, usporedbu ne možemo provesti u ALU (potrebno je predvidjeti poseban komparator)

bne \$rs,\$rt,pomak #  $PC \leftarrow PC + 4 + pomak$ , ako ( $r1 \neq r2$ )



I-tip

000101

rs

rt

<pomak>

Primjer (zakašnjelo grananje):

Standardni kôd koji na RISC arhitekturi ne bi bio ispravan:

```
move r4, r3  
move r1, r2  
jal x          ; bezuvjetni poziv potprograma, C je povratna adresa  
C: add r5, r5, 1
```

RISC prevodioc bi generirao:

```
move r4, r3  
move r1, r2  
jal x          ; bezuvjetni poziv potprograma, C je povratna adresa  
nop  
C: add r5, r5, 1
```

Optimirajući RISC prevodioc bi umjesto (neoptimirane) izvedbe:

```
move r4, r3  
move r1, r2  
jal x      ; bezuvjetno grananje  
nop  
C: add r5, r5, 1
```

... proizveo konačnu varijantu:

```
move r4, r3  
jal x      ; bezuvjetno grananje  
move r1, r2  
C: add r5, r5, 1
```

## Protočnost, sažetak

- moćan koncept koji omogućava višestruko povećanje performanse
- optimalna protočna struktura:
  - svaki segment odgovoran za približno jednak zahtjevan dio puta podataka
  - po jedna instrukcija izlazi iz cjevovoda u svakom periodu
  - prosječna performansa znatno bolja od neprotočne izvedbe
- nužne pretpostavke:
  - pažljivo odabran (ortogonalan) instrukcijski skup
  - μoperacije po segmentima jednak traju

## Protočnost, sažetak (2):

- protočnost omogućava efikasno iskorištavanje instrukcijskog paralelizma
- faktor ubrzanja  $\times 4$  i više!
- glavni izazov su hazardi:
  - tehnika prosljeđivanja obično pomaže (ali ne uvijek dovoljno)
  - zakašnjelo grananje i zakašnjelo učitavanje ponekad pomaže (prevodioc ne uspijeva uvijek naći zamjenu!)
  - hazardi će smetati još i više kod **superskalarnih** izvedbi

## Literatura

- S. Ribarić, *Arhitektura računala RISC i CISC*, Školska knjiga, Zagreb, 1996.
- D. A. Patterson, J. L. Hennessy, Computer Organization & Design, The Hardware/Software Interface, Morgan Kaufmann, 4th ed, 2009.

# Osnove grafičkih procesnih jedinica

uvod u ne-grafičke primjene GPU-ova

Siniša Šegvić  
UniZg-FER

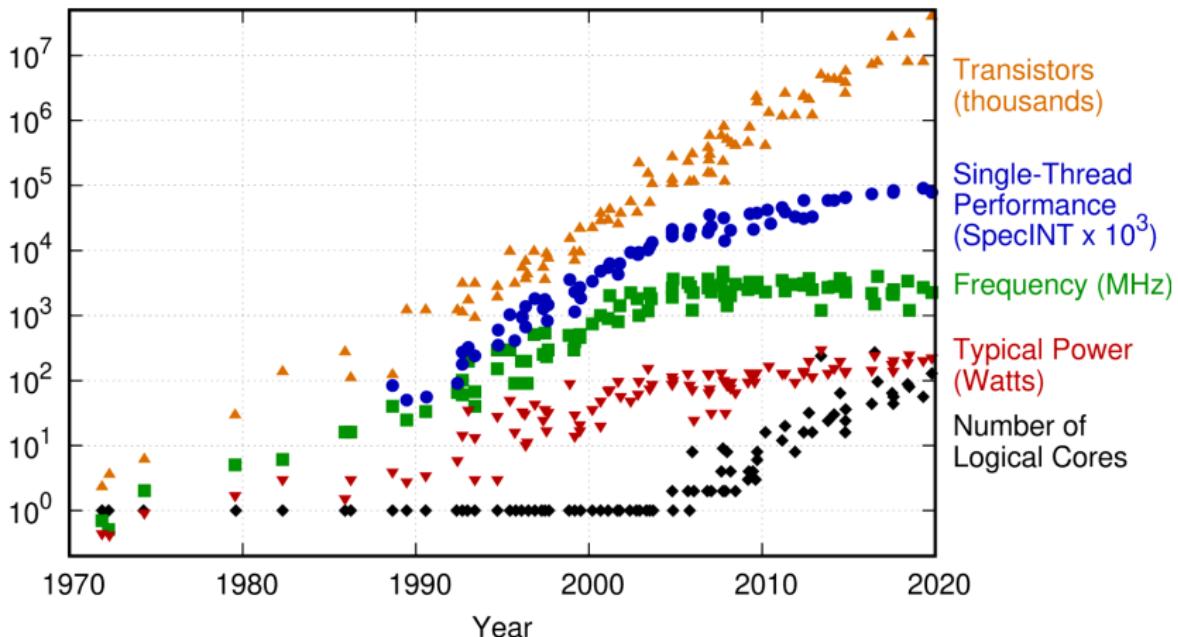
# PLAN

- usporedba s CPU-ovima
- organizacija GPU-ova: dretve, grupe, blokovi
- izražavanje GPU programa
- osnove OpenCL-a

# UVOD : SVIJET SE MIJENJA

Performansa skalarnih procesora ulazi u zasićenje

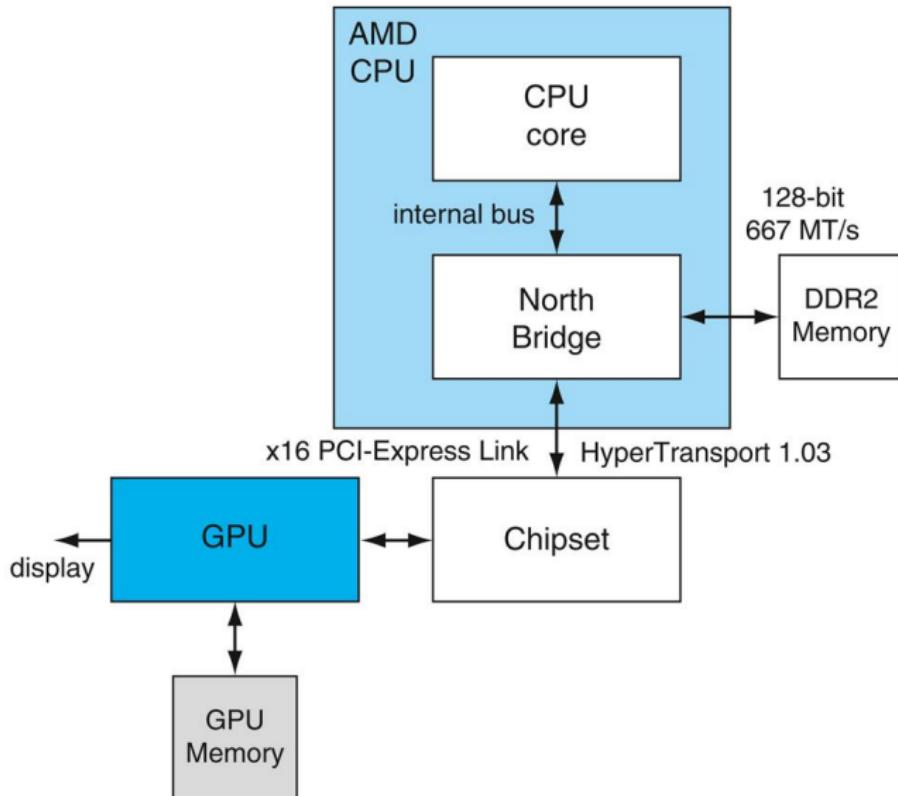
48 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2019 by K. Rupp

[rupp19github]

# GPU vs CPU : POZICIJA



[patterson20book]

## GPU vs CPU : VRŠNA PERFORMANSA (VP), FLOPS

CPU i9 10900K (Comet lake, 14nm, 206 mm<sup>2</sup>, 2020):

- 10 jezgara, 5.3 Ghz turbo
- svaka jezgra može izdati  $2 \times$  AVX-512 u svakom taktu
  - SIMD: jedna instrukcija obrađuje više parova podataka
  - AVX-512:  $512/32=16$  FMA operacija u fp32 preciznosti
- $\text{vp} = 10 * 5.3\text{G} * 16 * 2 * 2 = 3.4 \text{ TFLOPS}$

GPU A100 (Ampere, 7nm, 826 mm<sup>2</sup>, 2020, 40 GB RAM):

- 108 multiprocesora (SMP), 1.4 GHz
- SMP: 64 procesora (SP, CUDA core)
- $\text{vp} = 108 * 1.4\text{G} * 64 * 2 = 19.4 \text{ TFLOPS}$

## GPU vs CPU : VRŠNA PERFORMANSA (2)

Memorijski sustav (CPU):

- širina sabirnice (JEDEC DDR): 64 bit
- memorijska propusnost (DDR5 - 6000 MHz, 4 kanala): 192 GB/s

Memorijski sustav (GPU):

- širina sabirnice(A100): 5120 bit
- memorijska propusnost: 1.5 TB/s

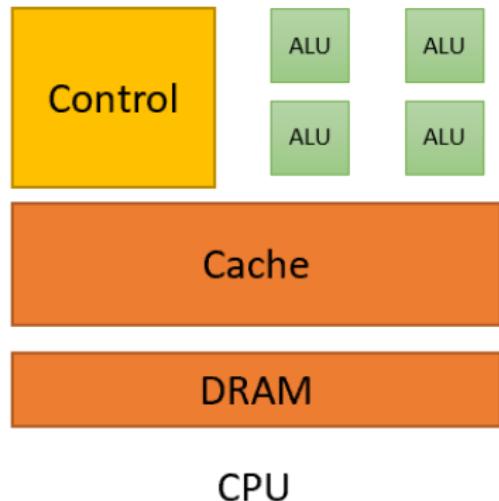
Vršna performansa je odokativan pokazatelj:

- GPU-ovi dolaze bliže vršnoj performansi od CPU-ova
  - CPU-ovi obavljaju raznovrsne zadatke s puno uvjetnih grananja
- za paralelne zadatke, GPU-ovi su  $50 \times$  brži
- za zadatke koji se ne mogu paralelizirati, GPU-ovi su neupotrebljivi

# GPU vs CPU : RASPODJELA RESURSA

Različita alokacija tranzistora:

- CPU - prvenstveno na priručne memorije i upravljanje
- GPU - prvenstveno na računanje



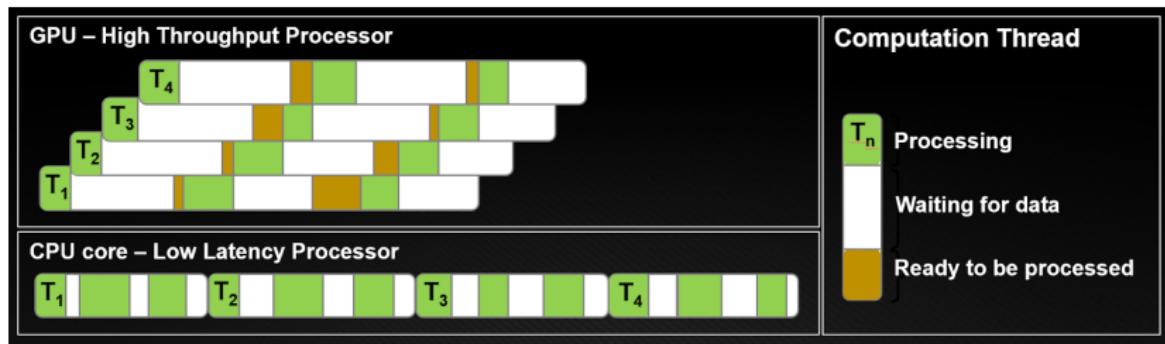
[gupta20nvidia]

# GPU vs CPU : PRISTUP LATENCIJI

Umanjivanje problema vezanih uz hazarde:

- CPU - skraćuje latenciju sofisticiranim cachevima
- GPU - održava propusnost sklopovskom višedretvenošću (SIMT)
  - ideja je slična razvijanju petlje (preplitanje nezavisnog koda)
  - za one koji žele naučiti malo više: SIMD < SIMT < SMT

<https://yosefk.com/blog SIMD-SIMT-SMT-parallelism-in-nvidia-gpus.html>



[gupta20nvidia]

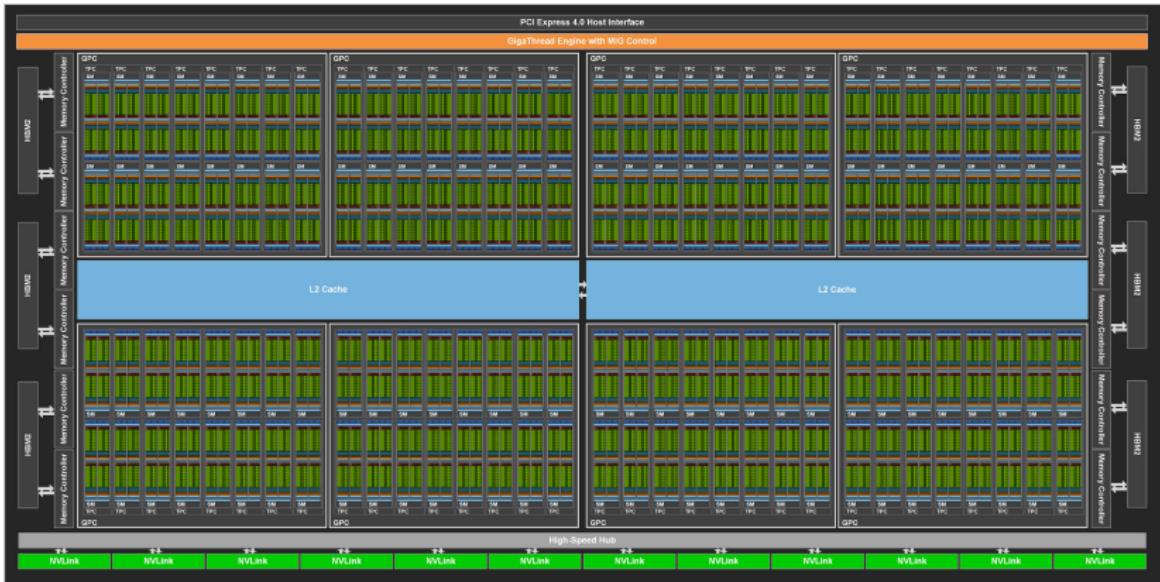
GPU uspijeva zaposliti procesore unatoč dužem čekanju na podatke

# GPU vs CPU : SAŽETAK

| Feature  | Multicore with SIMD | GPU             | i9-10900k | A100      |
|--|---------------------|-----------------|-----------|-----------|
| SIMD processors                                  | 8 to 32             | 15 to 128       | 10        | 108       |
| SIMD lanes/processor                             | 2 to 4              | 8 to 16         | 16        | 64        |
| Multithreading hardware support for SIMD threads | 2 to 4              | 16 to 32        | 2         | 32        |
| Largest cache size                               | 48 MiB              | 6 MiB           | 20 MiB    | 40 MiB    |
| Size of memory address                           | 64-bit              | 64-bit          | 64        | 5120      |
| Size of main memory                              | 64 GiB to 1024 GiB  | 4 GiB to 16 GiB | 128 GiB   | 40-80 GiB |
| Memory protection at level of page               | Yes                 | Yes             |           |           |
| Demand paging                                    | Yes                 | No              |           |           |
| Cache coherent                                   | Yes                 | No              |           |           |

[patterson20book]

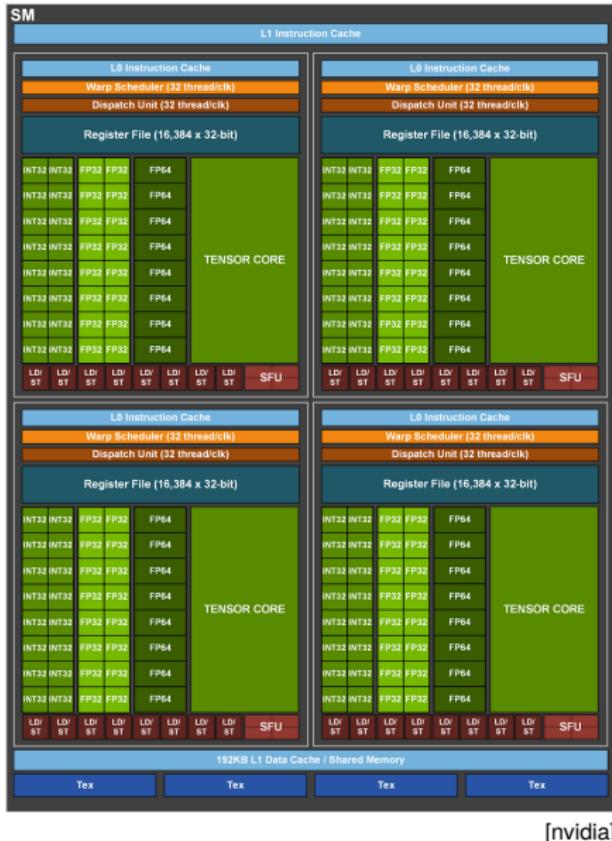
# GPU ORGANIZACIJA : CJELOKUPNI SUSTAV (A100)



[nvidia]

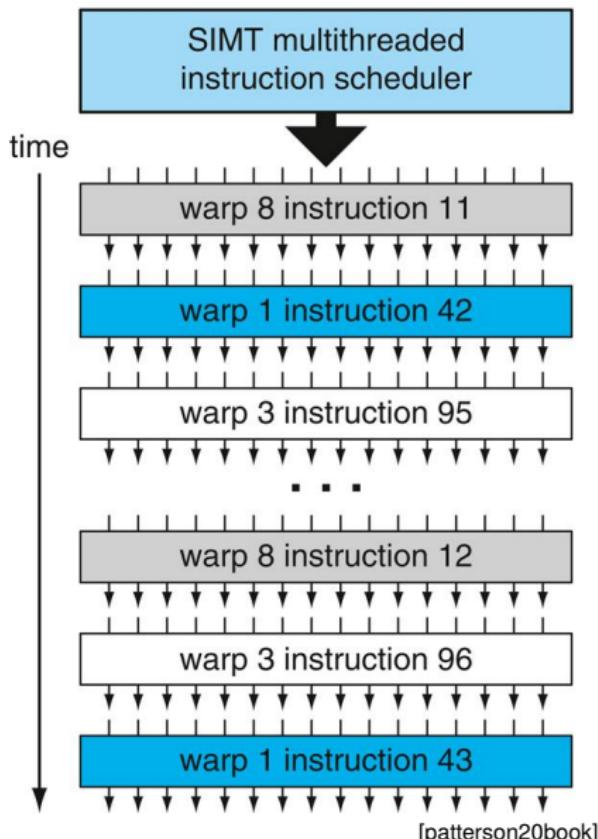
- 128 (108) multiprocesora (ugrubo, odgovaraju CPU jezgrama)
  - svaki multiprocesor ima 64 jednostavna procesora (CUDA core)
  - svi procesori multiprocesora izvršavaju istu instrukciju (SIMD-like)

# GPU ORGANIZACIJA : MULTIPROCESOR



- procesori (SP)  
multiprocesora (SMP)  
izvode različite dretve
- grupa (warp): skup dretvi  
koje se istovremeno  
izvršavaju na istom SMP-u
- dretve grupe su sinkrone:  
svaki SP izvršava istu  
instrukciju (dijeljeni PC!)
- grananje je dozvoljeno ali  
jako usporava izvođenje
- latencija se ublažava  
paralelnim izvođenjem **više  
grupa**

# GPU ORGANIZACIJA : SIMT



- SIMT: single instruction, multiple threads
- multiprocesor mijenja grupu nakon svakog takta
- **blok dretvi** (thread block): sve grupe dodijeljene istom SMP-u
- preklapanje grupa dretvi provodi se sklopovski
- srođno vektorskim (SIMD) i višedretventim računalima (SMT)
- možda najinovativniji koncept modernih GPU-ova

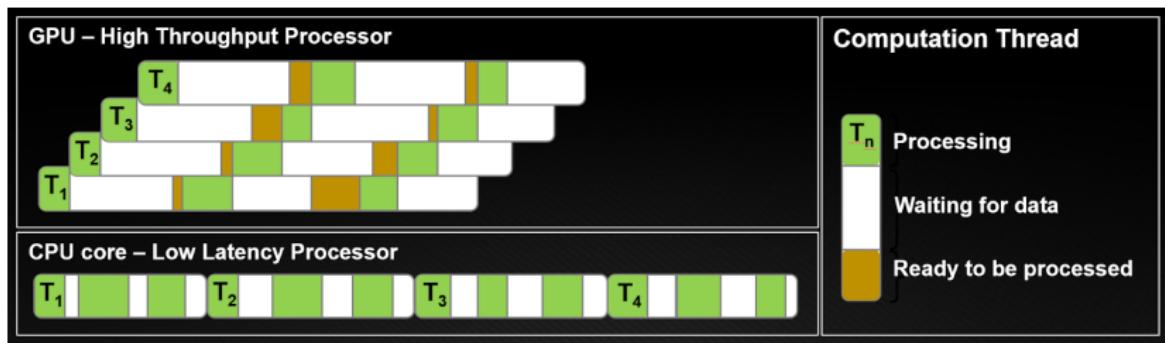
# GPU ORGANIZACIJA : PRISTUP LATENCIJI

Svaki procesor je protočan: ako imamo dovoljno grupa, možemo zamaskirati sporu priručnu memoriju i hazarde.

Zbog toga procesori najčešće uspijevaju raditi nešto korisno

Može biti korisno kad:

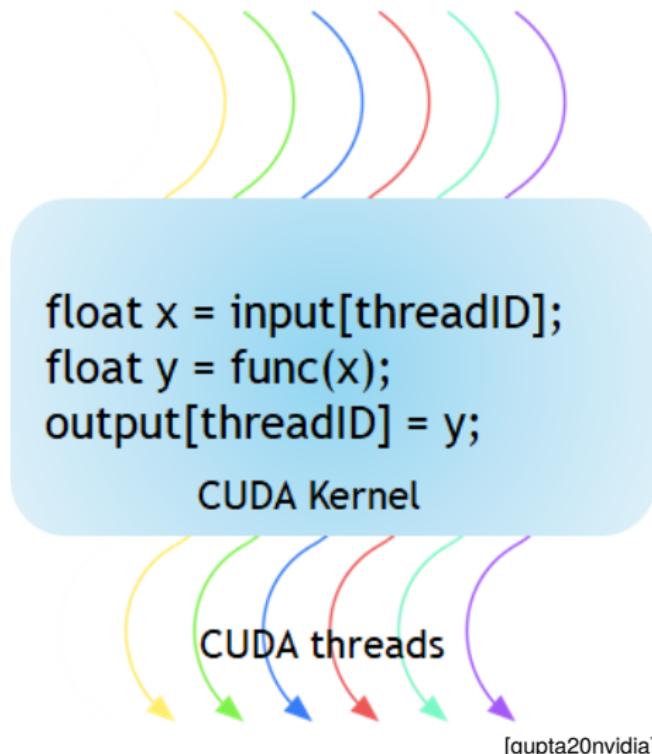
- na raspolaganju imamo puno grupa dretvi (tj. problem je paralelan)
- dretve ne koriste grananje



[gupta20nvidia]

# PROGRAMIRANJE : PROBLEMI

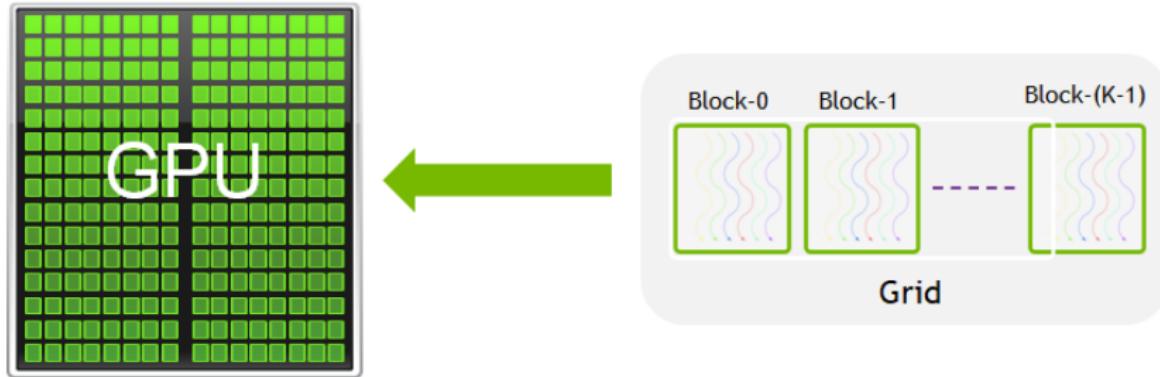
Pogodni problemi obrađuju mnogo podataka na sličan način:



# PROGRAMIRANJE : STRUKTURA

Programe za GPU izražavamo u obliku polja (grid) nezavisnih dretvi.

Programski okviri (npr. CUDA, OpenCL) transparentno grupiraju dretve u blokove te ih raspoređuju multiprocesorima

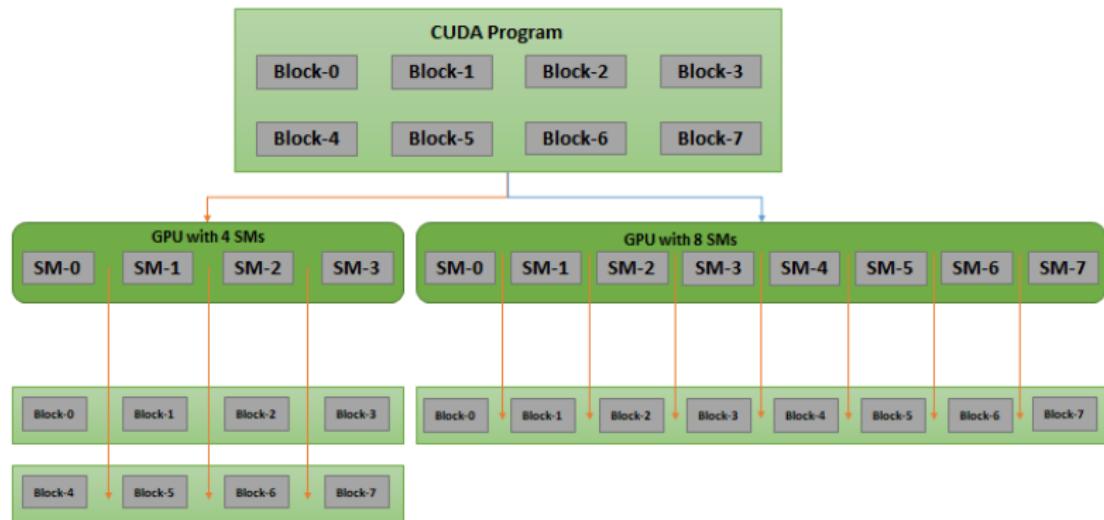


[gupta20nvidia]

Npr. ako jezgru za obradu floatova konfiguriramo tako da ima blokove od 256 dretvi, oni će na A100 biti raspoređeni u 4 grupe.

# PROGRAMIRANJE : STRUKTURA (2)

Prevedeni program može se pokretati na sklopoljtu različite snage:



[gupta20nvidia]

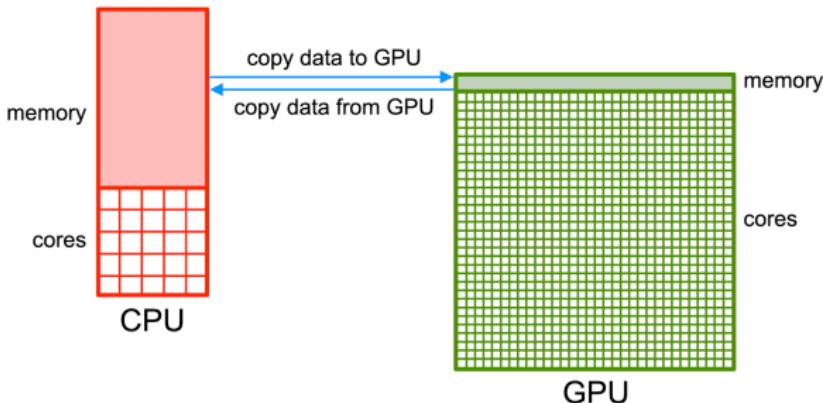
Ipak, najbolju performansu postižemo prevođenjem za ciljani uređaj

- korak dalje: ciljana optimizacija (TensorRT)

# PROGRAMIRANJE : IZVORNI KOD

Tipično, programi za GPU izvršavaju se barem djelomično na CPU-u

- CPU: učitavanje ulaza i pohranjivanje rezultata
- heterogeno računarstvo!



```
data = open("input.dat");
copyToGPU(data);
matrix_inverse(data.gpu);
copyFromGPU(data);
write(data, "output.dat");
# read the data on the CPU
# copy the data to the GPU
# perform a matrix operation on the GPU
# copy the resulting output to the CPU
# write the output to file on the CPU
```

# OPENCL: UVOD

OpenCL i CUDA su dominantni okviri za programiranje na GPU-ovima

Jezgra: osnovna jedinica koda za GPU (OpenCL, CUDA)

- sve dretve bloka su instance iste jezgre
- tipično odgovara jednoj funkciji

Prednosti OpenCL-a:

- podržava Intelove i AMD-ove GPU-ove (CUDA je NVidijin proizvod)
- može se izvršavati i na CPU-ovima

Instalacija na arch Linuxu:

```
$ sudo pacman -S intel-compute-runtime ocl-icd opencl-headers
```

## OPENCL: MINIMALNA JEZGRA

Ova jezgra izražava množenje jednog elementa cjelobrojnog polja zasebnom dretvom:

```
--kernel void simple_demo(  
    __global int *src,  
    __global int *dst,  
    int factor)  
{  
    int i = get_global_id(0);  
    dst[i] = src[i] * factor;  
}
```

Izvorni kod jezgre prevodimo pozivom funkcije `clBuildProgram`:

- na taj način osiguravamo just-in-time prevodenje za ciljani uređaj

Objekt koji enkapsulira jezgru dobivamo funkcijom `clCreateKernel`:

- simboličko ime jezgre (`simple_demo`) zadajemo argumentom

## OPENCL: IZVRŠAVANJE JEZGRE

Izvršavanje jezgre iniciramo funkcijom `c1EnqueueNDRangeKernel`:

- rezultat poziva: stvaranje polja dretvi (grid)
- svaka dretva - jedan prolazak kroz jegrenu funkciju

```
c1EnqueueNDRangeKernel(queue, kernel, 1,  
NULL, global_work_size, NULL, 0, NULL, NULL)
```

## OPENCL: IZVRŠAVANJE JEZGRE

Najvažniji parametri funkcije `c1EnqueueNDRangeKernel` su redom:

- `command_queue` - asinkroni red zadataka za odabrani uređaj
- `kernel` - jezgra koju želimo pokrenuti
- `work_dim` - broj dimenzija polja dretvi koje će instancirati jezgra
- `global_work_offset` - početni indeksi polja dretvi (ako NULL, onda se počinje od indeksa 0)
- `global_work_size` - polje od `work_dim` elemenata, određuje dimenzije polja dretvi
- `local_work_size` - polje od `work_dim` elemenata, određuje dimenzije blokova dretvi (ako NULL, blokove dimenzionira OpenCL)

<https://man.opencl.org/c1EnqueueNDRangeKernel.html>

## OPENCL: VJEŽBA

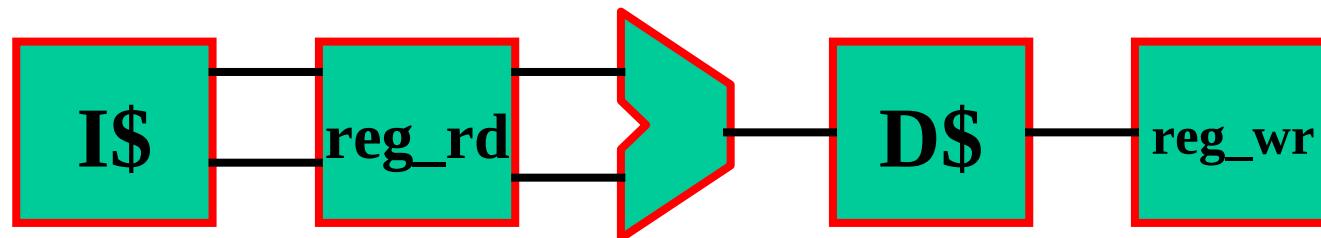
Četvrta laboratorijska vježba traži dvije naivne izvedbe množenja kvadratnih matrica:

- jednu u C-u za CPU,
- drugu u OpenCL-u za GPU.

Zadatak je izmjeriti brzinu izvođenja dviju izvedbi, kao i vrijeme prijenosa podataka s CPU-a na GPU.

Upute sadrže više pitanja koja omogućavaju samotestiranje: nemojte ih preskočiti!

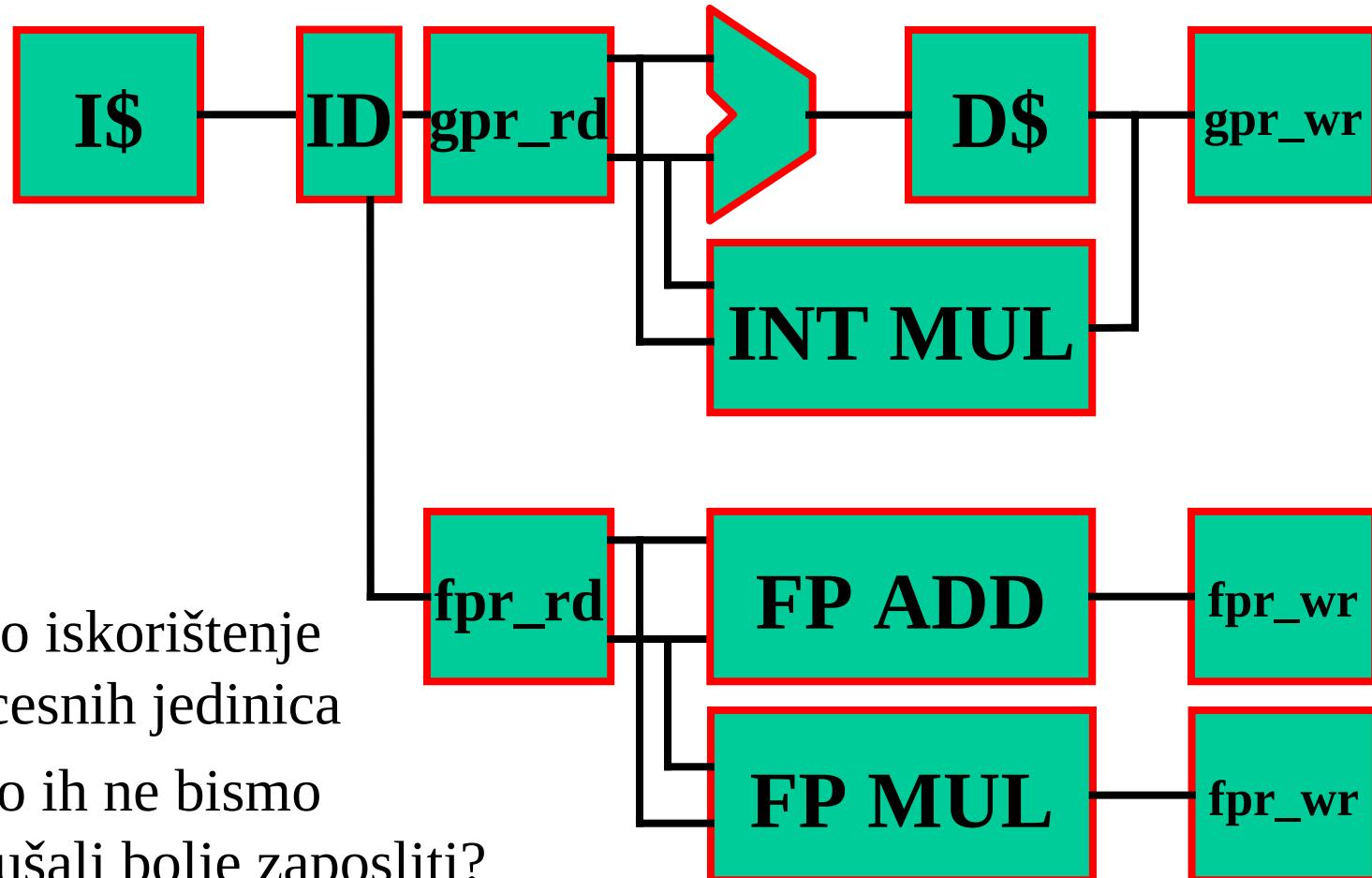
# Od protočnosti do višestrukog izdavanja



- temeljni put podataka arhitekture MIPS ne podržava ni množenje ni dijeljenje ni operacije s pomičnim zarezom
- štednja na broju tranzistora nije opravdana zbog Mooreovog zakona
- logično proširenje arhitekture: dodati višestruke procesne jedinice i uklopliti ih u temeljni cjevovod

## Složena protočna organizacija s jednostrukim izdavanjem:

- više protočnih grana koje rade usporedno
- različite protočne grane imaju različite latencije
- veće mogućnosti bez velikog usporenja temeljnih operacija

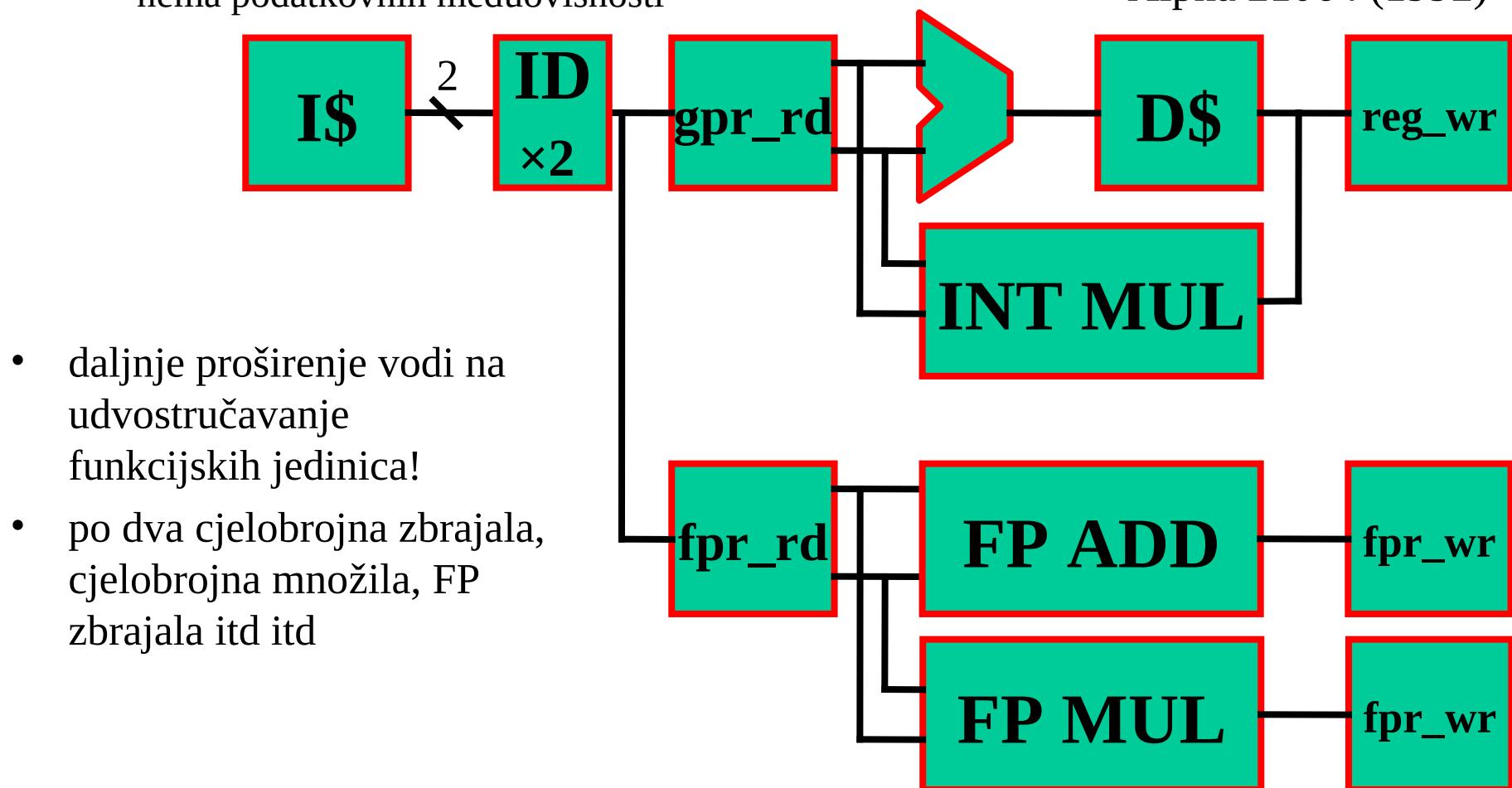


# Koncept višestrukog izdavanja instrukcija:

- CPU istovremeno dohvaća i dekodira dvije instrukcije!
- instrukcije se izvode **usporedno**, ako:
  - koriste različite resurse,
  - prva instrukcija nije uvjetno grananje
  - nema podatkovnih međuvisnosti

Predstavnici:

- MIPS 4000 (1991)
- Alpha 21064 (1992)



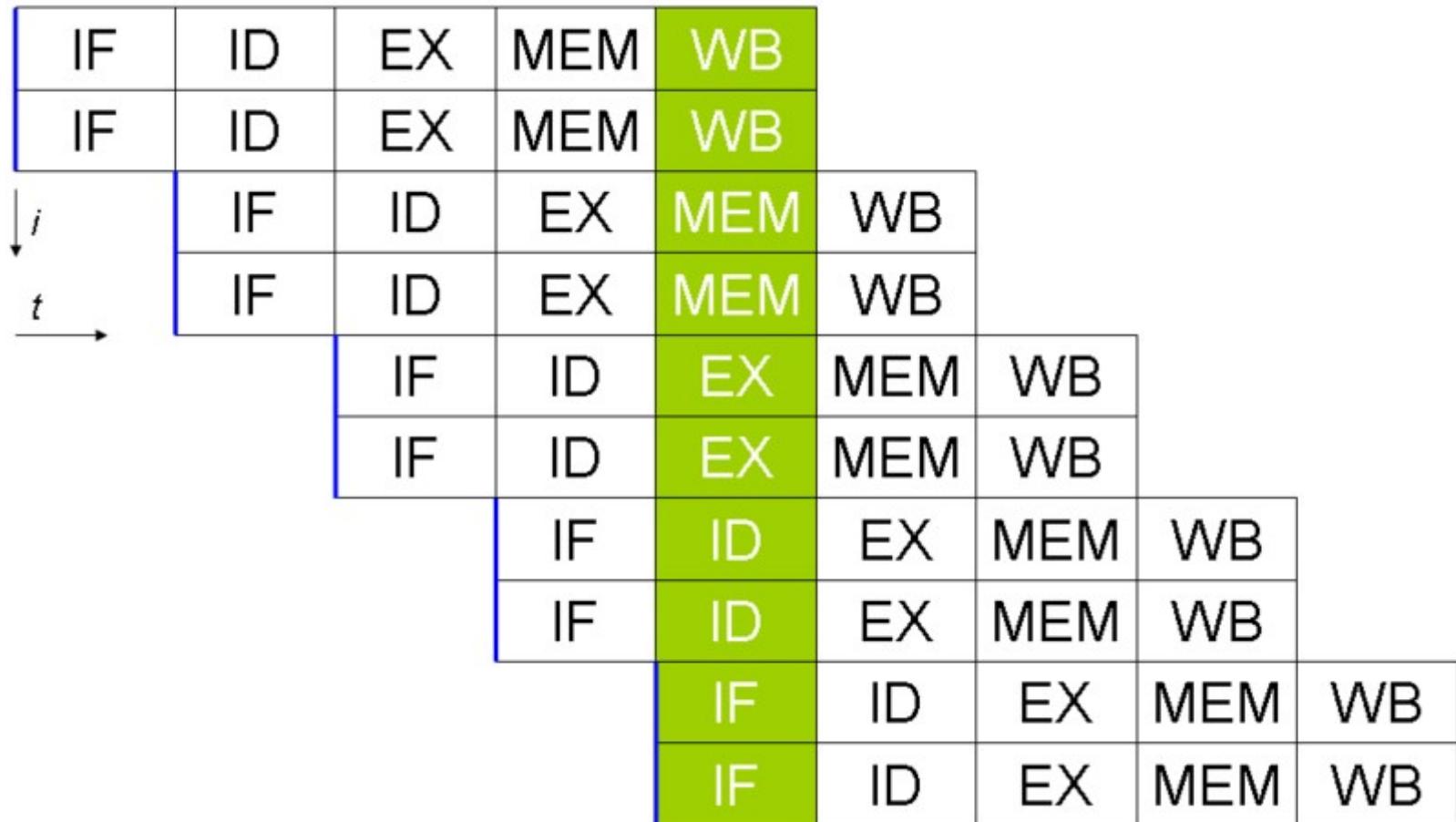
**Ideja:** istovremenim **izdavanjem** više instrukcija razotkriti više instrukcijskog paralelizma nego što to omogućava protočnost

- u implementaciji: replicirane protočne funkcijalne jedinice!
- u svakom ciklusu (pokušati) započeti s obradom više instrukcija
- uz n-terostruko izdavanje (n-way multiple issue) vršni CPI = 1/n

Koncept u načelu jednostavan, ali se implementacija komplicira:

- redoslijed instrukcija i hazardi onemogućavaju iskorištenje resursa
- cijena mjehurića može biti veća nego kod skalarnog CPU
- slabo skaliranje zbog izvedbenih problema  
(m ... br. protočnih jedinica):
  1. broj sabirnica za čitanje registarskog skupa:  $2 \times m$
  2. broj ad-hoc veza za prosljeđivanje:  $m$  ili  $m^2$
- u praksi (Haswell), trenutno se postiže CPI=0.5
  - $n=4, m=8, 14-19$  protočnih segmenata

Gantov dijagram procesora s dvostrukim izdavanjem, u idealnom slučaju (bez hazarda):



Dva načina raspoređivanja (engl. dispatch) kod procesora s višestrukim izdavanjem:

## 1. **statičko raspoređivanje** (VLIW)

- prevoditelj grupira instrukcije koje se izdaju zajedno
- **pakete** instrukcija promatramo kao “duge instrukcije”
- prevoditelj detektira hazarde i po potrebi umeće nop-ove

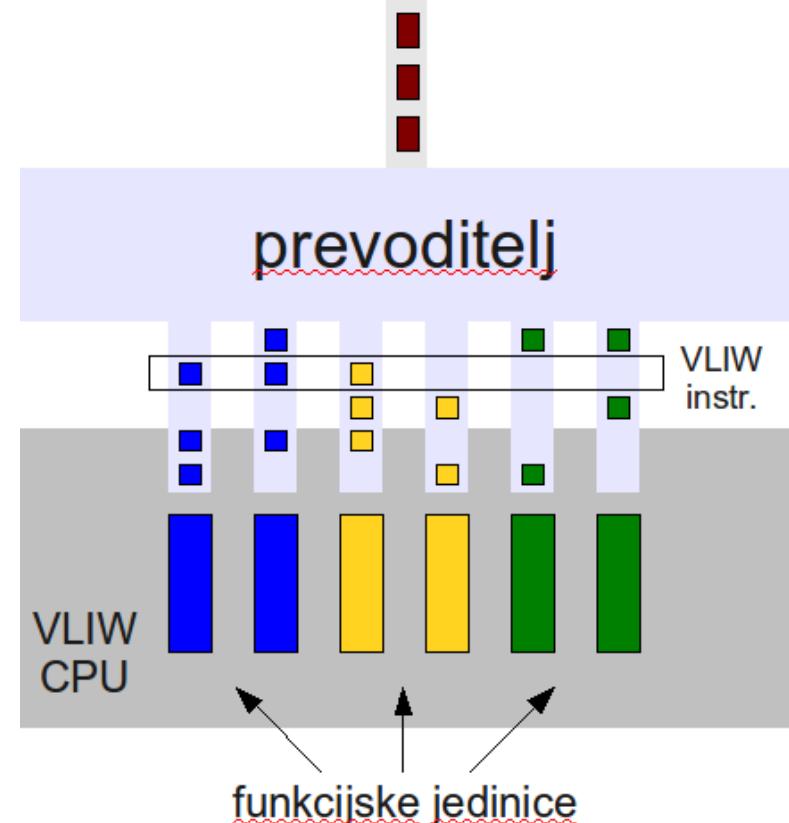
## 2. **dinamičko raspoređivanje** (“superskalarni” RISC)

- procesor analizira instrukcijski tok i dinamički odabire instrukcije koje će se izdati u sljedećem ciklusu
- procesor dinamički otkriva i razrješava hazarde (koncept upravljanja temeljenog na toku podataka)
- prevoditelj pomaže prikladnim razmještajem instrukcija

# Višestruko izdavanje sa statičkim raspoređivanjem (statičko višestruko izdavanje)

- instrukcije se grupiraju u **pakete**
  - paket: grupa instrukcija koja se može izdati u jednom ciklusu
  - prevoditelj treba poznavati resurse procesora
- VLIW: paket = duga instrukcija
  - definira operacije koje se mogu izvoditi konkurentno
- posao prevoditelja:
  - organizirati instrukcije u pakete
  - bez međuvisnosti unutar paketa!
  - popuniti neiskorištene okvire (nop)

```
int main(int argc, char** argv){  
    for (int i=0; i<argc; ++i){  
        // ...  
    }  
}
```



# MIPS sa statičkim dvostrukim izdavanjem

- jedna instrukcija tipa ALU/branch
- jedna instrukcija tipa load/store
- paket (svežanj) poravnat na granici 64-bitne riječi:
  - prvo ALU/branch, onda load/store
  - neiskorištene okvire punimo instrukcijama nop

| adresa | vrsta instrukcije | protočni segmenti |    |    |     |     |     |    |
|--------|-------------------|-------------------|----|----|-----|-----|-----|----|
|        |                   | IF                | ID | EX | MEM | WB  |     |    |
| n      | ALU/branch        |                   |    |    |     |     |     |    |
| n + 4  | Load/store        | IF                | ID | EX | MEM | WB  |     |    |
| n + 8  | ALU/branch        |                   | IF | ID | EX  | MEM | WB  |    |
| n + 12 | Load/store        |                   | IF | ID | EX  | MEM | WB  |    |
| n + 16 | ALU/branch        |                   |    | IF | ID  | EX  | MEM | WB |
| n + 20 | Load/store        |                   |    | IF | ID  | EX  | MEM | WB |

# Hazardi pri dvostrukom izdavanju

- podatkovni RAW hazard segmenta EX
  - u skalarnoj izvedbi, prosljeđivanje otklanja sve zastoje
  - sada rezultat ALU ne možemo koristiti u istom paketu:
    - add \$r1, \$r2, \$r3
    - load \$r4, 0(\$r1)
    - instrukcije potrebno razdvojiti u dva paketa (efektivno, imamo zastoj)
- podatkovni RAW hazard segmenta MEM
  - efekt instrukcije load sad je vidljiv nakon 2 instrukcije!
- više instrukcija se izvodi usporedno  $\Rightarrow$  više hazarda!
  - potreba za agresivnim raspoređivanjem još izraženija!

## Primjer: algoritam nad cjelobrojnim poljem

```
void addScalar(int* p, int scalar, int* limit){  
    while (p!=limit){  
        *p+=scalar;  
        ++p;  
    }  
}
```

Razmatramo izvedbu tijela funkcije za MIPS s dvostrukim izdavanjem gdje vrijedi sljedeće:

- uvedena je automatska detekcija hazarda (MIPS II nadalje):
  - nema potrebe za eksplicitnim nop-om nakon instrukcije lw
- uvedeno je sklopolje za predviđanje grananja koje omogućava pravovremeno pribavljanje parova instrukcija na odredišnoj adresi
  - zakašnjelo grananje više ne pomaže zbog dvostrukog izdavanja (bez predviđanja grananja trebala bi nam 3 priključka za kašnjenje!)
  - novije MIPS arhitekture ipak zakašnjelo granaju zbog kompatibilnosti

Pogledajmo prvo kako bi izgledao neoptimirani strojni kod ako prepostavimo sljedeći raspored registara:

- \$s1 ... int\* p
- \$s2 ... int scalar
- \$s3 ... int\* limit

```
...
beq  $s1, $s3, Exit # $s3 .. int* limit
nop
```

Loop:

```
lw    $t0, 0($s1)      # $s1 .. int *p
addu $t0, $t0, $s2      # $s2 .. int scalar
sw    $t0, 0($s1)      #
addi $s1, $s1,4        #
bne   $s1, $s3, Loop
nop
```

Exit:

```
...
```

# Raspoređivanje neoptimiranog koda na računalu s dvostrukim izdavanjem:

|        | ALU/branch            | Load/store       | ciklus |
|--------|-----------------------|------------------|--------|
| Loop : | nop                   | lw \$t0, 0(\$s1) | 1      |
|        | nop                   | nop              | 2      |
|        | addu \$t0, \$t0, \$s2 | nop              | 3      |
|        | addi \$s1, \$s1, 4    | sw \$t0, 0(\$s1) | 4      |
|        | bne \$s1, \$s3, Loop  | nop              | 5      |

Zamijenili smo redoslijed instrukcija addi i sw da postignemo bolju popunjenošću protičnih jedinica.

CPI = 5/5 = 1 (samo malo bolje od jednostrukog izdavanja!)

# Optimirani strojni kod (MIPS, 2× staticko izdavanje)

```
Loop: lw    $t0, 0($s1)
      addi $s1, $s1,4
      addu $t0, $t0, $s2
      bne $s1, $s3, Loop
      sw   $t0, -4($s1)
```

## Razlike:

- instrukciju addi smjestili smo prije instrukcije addu koja ne koristi njen rezultat
- tu izmjenu smo kompenzirali u instrukciji sw navođenjem pomaka -4
- instrukciju sw smo smjestili u priključak za kašnjenje instrukcije bne

# Raspoređivanje optimiranog koda tijekom izvođenja:

|        | ALU/branch              | Load/store         | ciklus |
|--------|-------------------------|--------------------|--------|
| Loop : | nop                     | lw \$t0 , 0(\$s1)  | 1      |
|        | addi \$s1 , \$s1 , 4    | nop                | 2      |
|        | addu \$t0 , \$t0 , \$s2 | nop                | 3      |
|        | bne \$s1 , \$s3 , Loop  | sw \$t0 , -4(\$s1) | 4      |

CPI = 4/5 = 0.8 (bolje nego prije, ali slabije od vršnog CPI = 0.5)

$n_i = 5/\text{iteraciji}$

[Patterson08]

Prevoditelj može pomoći razvijanjem petlje (loop unroll):

|       | ALU/branch            | Load/store         | ciklus |
|-------|-----------------------|--------------------|--------|
| Loop: | addi \$s1, \$s1, 16   | lw \$t0, 0(\$s1)   | 1      |
|       | nop                   | lw \$t1, -12(\$s1) | 2      |
|       | addu \$t0, \$t0, \$s2 | lw \$t2, -8(\$s1)  | 3      |
|       | addu \$t1, \$t1, \$s2 | lw \$t3, -4(\$s1)  | 4      |
|       | addu \$t2, \$t2, \$s2 | sw \$t0, -16(\$s1) | 5      |
|       | addu \$t3, \$t3, \$s2 | sw \$t1, -12(\$s1) | 6      |
|       | nop                   | sw \$t2, -8(\$s1)  | 7      |
|       | bne \$s1, \$s3, Loop  | sw \$t3, -4(\$s1)  | 8      |

$$CPI = 8/14 = 0.6 \text{ (vršni CPI = 0.5)}$$

[Patterson08]

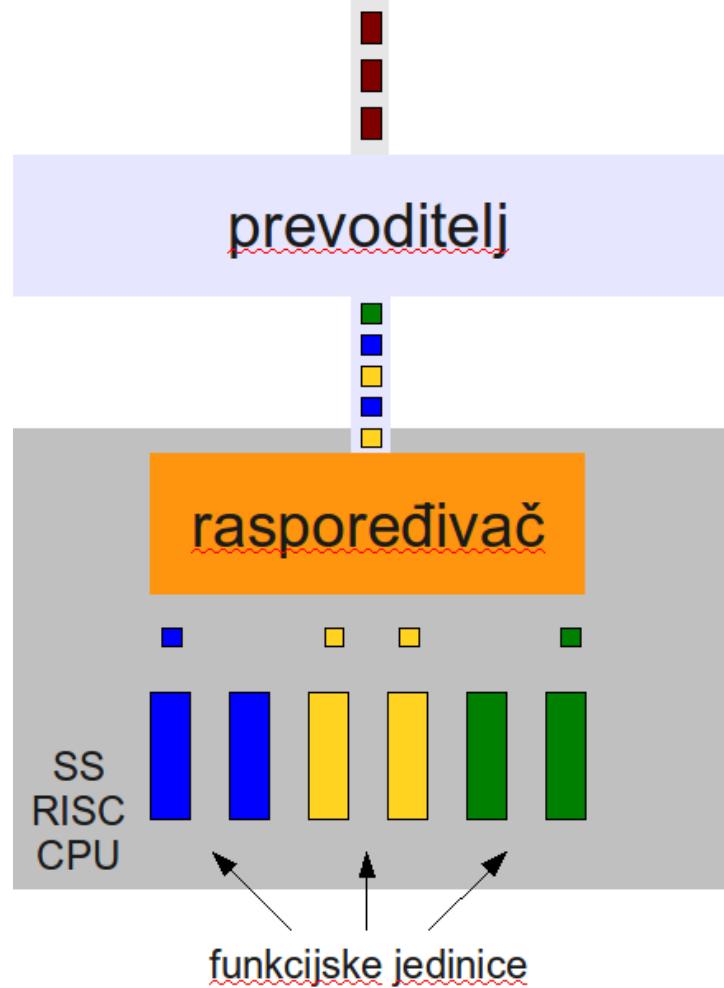
$$n_i = 3.5/\text{iteraciji}$$

**Cijena:** veći potrošak registara, duži i složeniji kod

# Dinamičko višestruko izdavanje

- “superskalarni” procesori
  - dinamički odlučuju hoće li izdati 0, ili 1, ili 2, ili više instrukcija
  - čuvaju semantiku programa
- manji značaj prevoditelja:
  - prevoditelj ne generira kod koji je strogo prilagođen jednoj arhitekturi
  - naravno, to ne znači da pomoć prevoditelja nije dragocjena
- potrebni dodatni koncepti:
  - izvođenje izvan redosljeda
  - preimenovanje registara
  - predviđanje grananja
  - spekulativno izvođenje

```
int main(int argc, char** argv){  
    for (int i=0; i<argc; ++i){  
        // ...  
    }  
}
```



## Izvođenje izvan redosljeda, dinamičko raspoređivanje

Redosljed instrukcija kao prepreka instrukcijskom paralelizmu:

```
div.d f0, f2, f4  
add.d f10, f0, f8 # <- podatkovni hazard vrste RAW  
sub.d f12, f8, f14 # <- ovdje nema podatkovnog hazarda,  
# ali svejedno moramo čekati
```

Instrukcija add mora čekati dugu instrukciju div (hazard RAW)

Instrukcija sub čeka instrukciju add iako nema podatkovnih ovisnosti:

- redosljed je svojevrsni strukturni hazard kod statičkog raspoređivanja
- nema podatkovnih hazarda za instrukciju sub!
- ako želimo bolje iskoristiti postojeći ILP, moramo omogućiti izvođenje instrukcija **izvan redosljeda**
- **dinamičko raspoređivanje** implicira izdavanje instrukcija koje nemaju:
  - strukturne hazarde (imaju slobodnu odgovarajuću funkciju jedinicu)
  - podatkovne RAW hazarde

Ali, zašto prevoditelj nije stavio sub.d **prije** div.d?

- zato što se radilo o školskom primjeru ☺
- idemo sada pogledati jedan realni primjer:

```
lw f14, 30(r6)
mul f0, f8,f0
sub.d f12, f8,f14 #dilema: redoslijed sub i add!
add.d f10, f0,f10
```

- stvarna latencija instrukcije lw nije poznata u trenutku prevodenja programa
  - od 1 ciklus (L1), 4 ciklusa (L2), 16 ciklusa (L3), do 100 ciklusa (RAM) ili 1e6 ciklusa (disk)
  - prevoditelj **ne zna** hoće li prije završiti lw ili mul
- ⇒ najbolje rezultate postiže dinamičko raspoređivanje!

# Motivacija za dinamičko raspoređivanje

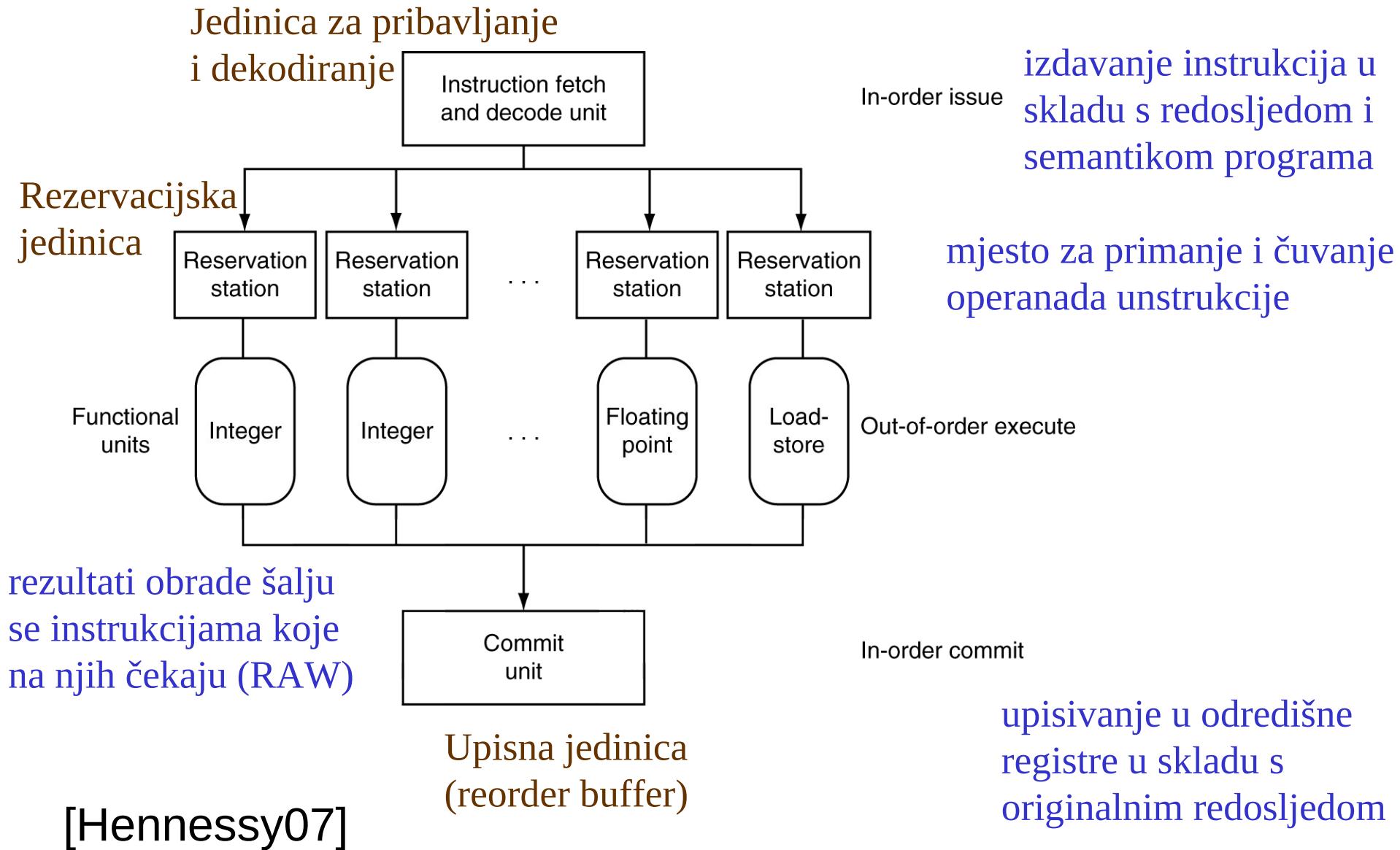
Registarski RAW hazardi mogu biti znatno ublaženi marljivom optimizacijom (redoslijed instrukcija određen statičkom analizom)

Međutim, sve hazarde nije moguće predvidjeti statičkom analizom:

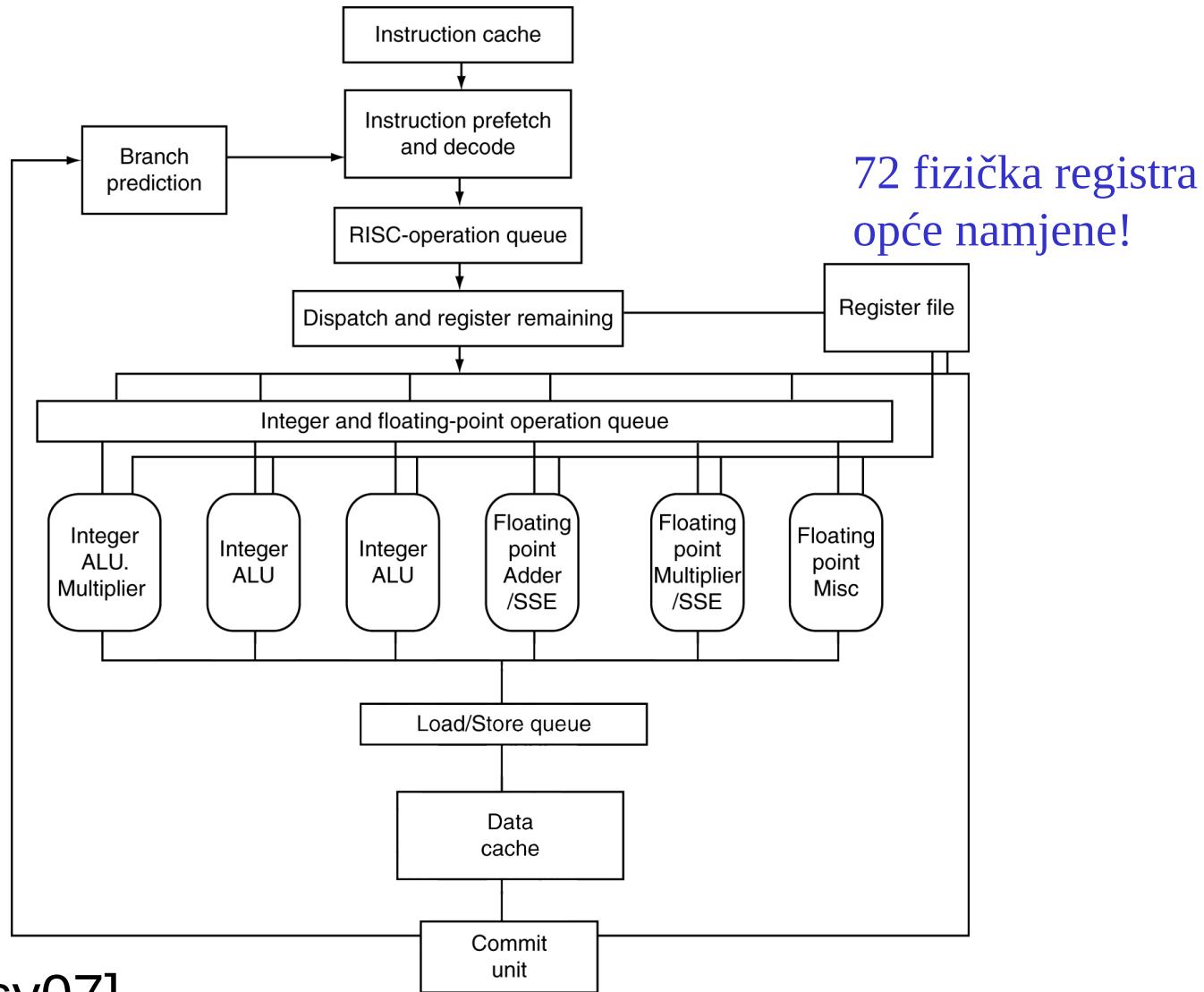
- pogotovo memorijske hazarde, pogotovo u superskalarnom procesoru
- ishod uvjetnog grananja također nije poznat tijekom prevodenja (doduše, profiliranjem možemo doznati statistička svojstva ishoda)
- latencija instrukcija može se razlikovati na različitim izvedbama ISA-e

Stoga je koncept izvođenja izvan redoslijeda danas prisutan kod većine procesora opće namjene, problemima unatoč (iznimno složena izvedba upravljanja, problematične iznimke, ...)

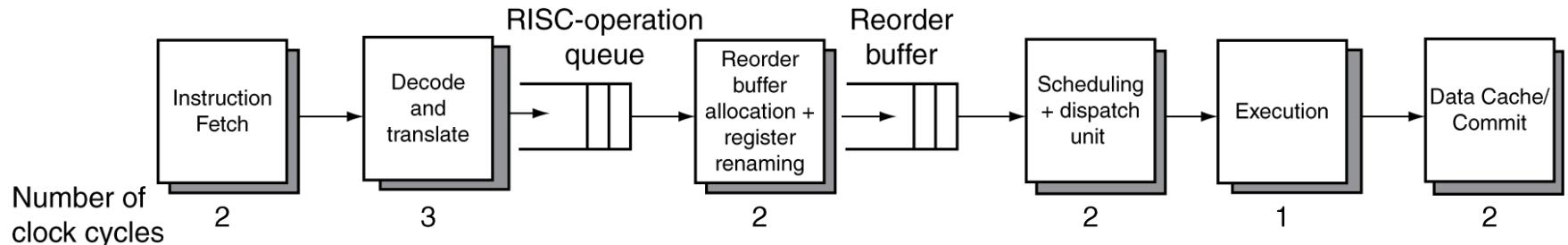
# Suvremeni CPU-a s dinamičkim raspoređivanjem



# Organizacija procesora Opteron X4



# Protočna struktura procesora Opteron X4



[Hennessy07]

- do sveukupno 106 aktivnih RISC instrukcija
- Usko grlo performanse:
  - međuovisnosti instrukcija
  - krivo predviđeno grananje
  - kašnjenje memorijskog pristupa

## Problemi vezani uz dinamičko raspoređivanje:

- moramo imati puno veću procesnu moć (broj procesnih jedinica) od ciljanog efektivnog CPI-a (0.5)
- moramo imati moćni prednji kraj koji je u stanju brzo opslužiti instrukcijama granu(e) puta podataka bez hazarda
- ograničen broj registara programskog modela je problematičan (WAR,WAW)
  - vrijedi i za nove arhitekture, gdje je broj registara ograničen širinom instrukcije
  - pogotovo vrijedi za CISC arhitekture (x86 ima 8 GPR)
- uvjetno grananje, promašaji priručne memorije?

# Zaključak: performansa izvođenja slijednih algoritama ulazi u zasićenje

- Agresivniji pristupi korištenja ILP-a (superskalarnost itd.) pomažu, ali njihova primjena dovodi do slabijeg iskorištenja resursa:
  - tranzistori (površina integriranog sklopa)
  - energija
  - mogućnosti razvojnog odjela
- Čini se da će izvođenje izvan redosljeda ostati
  - pogotovo u svijetu x86 (uz obaveznu štednju energije)
    - manja ovisnost o optimiziranom prevodiocu
    - bolje ponašanje u prisutnosti nepredviđenih zastoja
- Ipak, i dalje postoje procesori koji izvode unutar redosljeda
  - Intel Itanium (VLIW) oslanja se na statičku optimizaciju
  - IBM Power6, Intel Atom

# MIMD, SIMD vs. SISD+ILP

- stari vic koji **možda** više neće biti smiješan:  
“paralelno računarstvo je tehnologija budućnosti... ...i uvijek će to biti”
  - SIMD GPGPU ulazi na mala vrata (Sh, Cg, CUDA)
    - STI Cell, Ati, Nvidia, Intel Larrabee
  - značajan napredak na području MIMD
    - višejezgrena barijera probijena, međutim broj jezgri još uvijek relativno mali
    - procesori s tisuću jezgara tehnološki mogući, nema programske podrške
  - najčešći programi su još uvijek slijedni,
    - nije realno očekivati da će se to brzo promijeniti...
    - fokus inovacije: programske paradigme za MIMD (posebno **implicitne**)
- npr, usporedno računanje linearne kombinacije pomoću OpenMP-a:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++){
    sum = sum + myfun(i)*a[i];
}
```

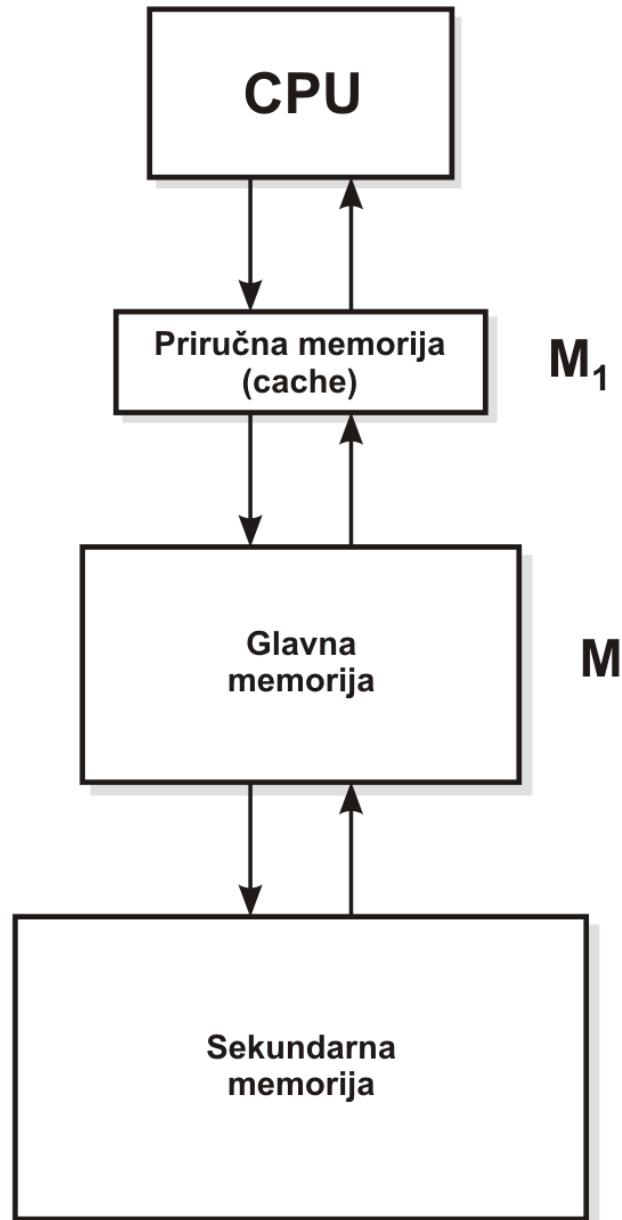
# **VIRTUALNI MEMORIJSKI SUSTAV**

- 1. Memorijска хијерархија**
2. Fizički i logički adresni prostor
3. Straničenje
4. Višerazinsko straničenje, translacijski spremnik
5. Segmentacija
6. Detalji izvedbe

## Prisjetimo se memorejske hijerarhije

- niže razine imaju **veći kapacitet** i **veću latenciju**
- ideja: smanjiti prosječnu latenciju korištenjem lokalnih kopija “popularnih” podataka u bržoj memoriji
- želimo da, prividno, računalo ima kapacitet diska i brzinu registara

|                  |  |
|------------------|--|
| zaporni sklopovi | .05 ns ( $0.2 \times \Delta T_{CPU}$ ), 100B |
| registri (SRAM)  | .25 ns ( $1 \times \Delta T_{CPU}$ ), 500B   |
| cache (SRAM)     | 1 ns ( $4 \times \Delta T_{CPU}$ ), 64kB     |
| RAM (DRAM)       | 50 ns, 1 GB                                  |
| diskovi          | 10 ms, 1 TB                                  |



Memorijska hijerarhija:

- $(M_1, M_2, M_3, \dots M_n)$
- $M_i$  “podređena” memoriji  $M_{i-1}$
- CPU izravno komunicira s  $M_1$

Neka je zadano:

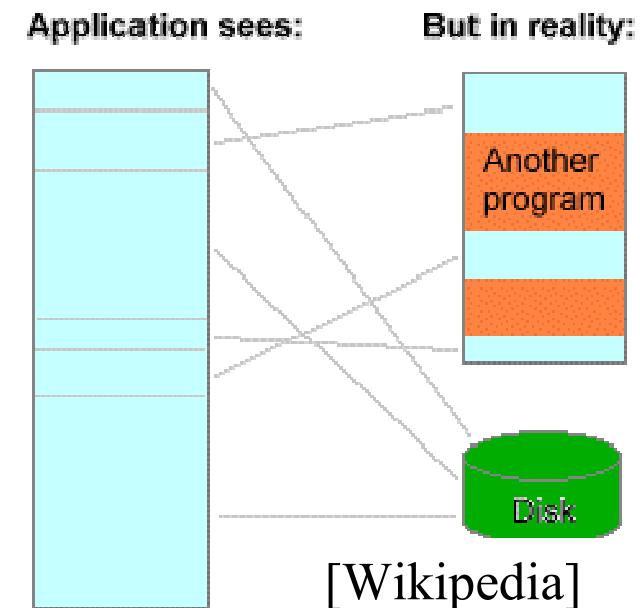
$$\begin{aligned}
 c_i &- \text{cijena po bitu} \\
 t_{ai} &- \text{vrijeme pristupa} \\
 S_i &- \text{kapacitet memorije}
 \end{aligned}$$

Tada vrijedi:

$$\begin{aligned}
 c_i &> c_{i+1} \\
 t_{ai} &< t_{ai+1} \\
 S_i &< S_{i+1}
 \end{aligned}$$

## Virtualna memorija, **glavne ideje**:

- **Virtualno** proširiti kapacitet radne memorije (DRAM) korištenjem prostora na magnetskom disku
  - **virtualna memorija** brzine DRAM-a, a veličine diska
  - kao i kod **priručnih memorija**, pouzdajemo se u **lokalnost pristupa**
- Dodatne funkcije (ne manje važne):
  - **adresno preslikavanje**: programima (procesima) pružiti **linearan** pogled na diskontinuirani fizički prostor
  - **transparentno** dijeljenje memorije
  - **zaštita pristupa**: onemogućiti neželjenu interakciju među procesima (korisničkim i jezgrenim)
    - posebno važno na višeprogramske računalima!



D.J. Wheeler: "Any problem in computer science can be solved with another layer of indirection. But that usually will create another problem."

# VIRTUALNI MEMORIJSKI SUSTAV

1. Memorijska hijerarhija
2. **Fizički i logički adresni prostor**
3. Adresno preslikavanje
4. Višerazinsko straničenje, translacijski spremnik
5. Segmentacija
6. Detalji izvedbe

## Fizički adresni prostor

Skup stvarnih memorijskih lokacija oblikuje **fizičku memoriju**.

Funkciju fizičke memorije obavljaju elektronički uređaji priključeni na sabirnicu procesora, odnosno računala.

Adrese koje su jednoznačno dodijeljene fizičkim memorijskim lokacijama čine **fizički adresni prostor FAP**.

Npr, za 32-bitno računalo s 512 MB memorije:  $|FAP|=5.4e8B$ .

## Logički adresni prostor

Logički adresni prostor LAP je skup svih adresa koje generira programski model procesora.

Adrese varijabli i potprograma te pokazivači stogova su logičke adrese.

Npr, za 32-bitno računalo s 512 MB memorije,  $|LAP| = 4.3e9$  B

## Mogući odnosi između veličina fizičkog i logičkog prostora:

$|LAP| = |FAP|$ : često kod računala s 16-bitnim adresama

$|LAP| < |FAP|$

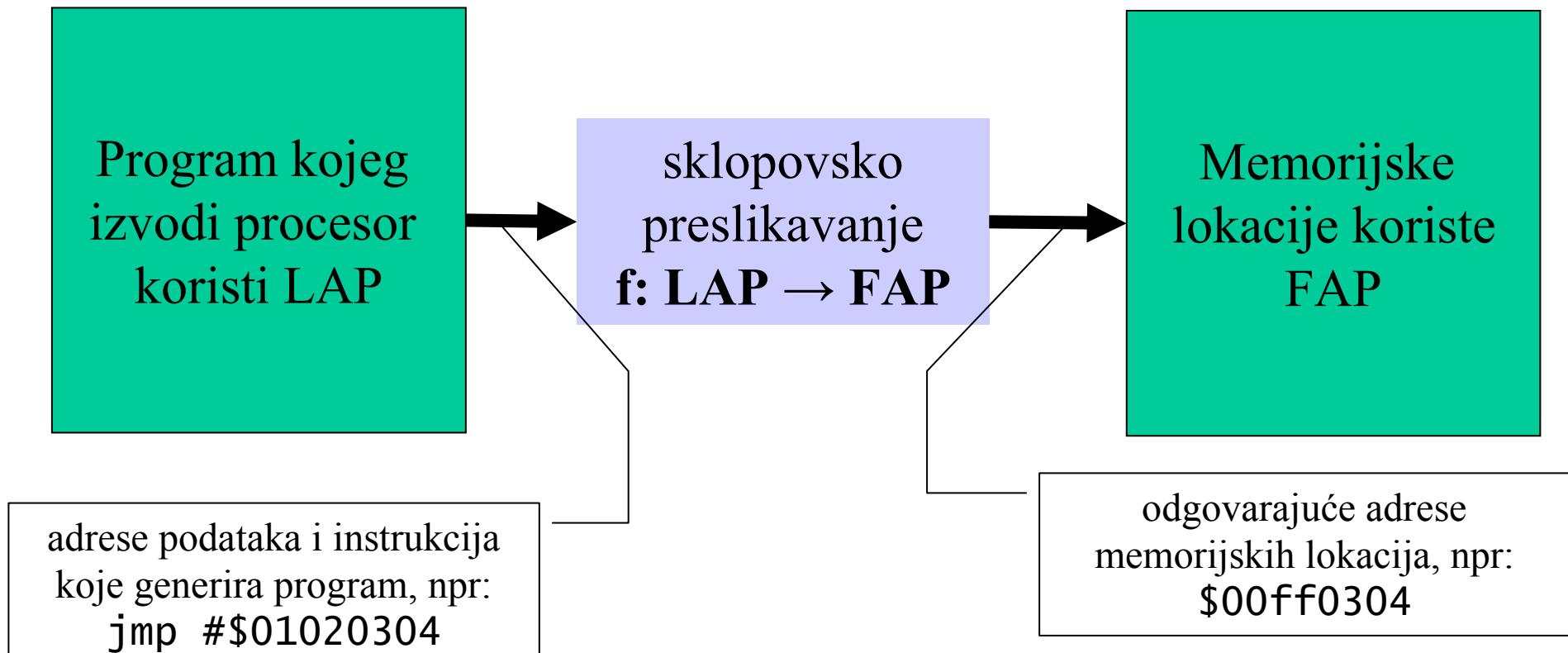
- kod računala bez adresnog preslikavanja:
  - FAP se izvodi s prekrivanjem (overlay)
  - programer odabire "vidljive" memorejske sklopove
- kod nekih 32-bitnih računala (x86: PAE):
  - procesor ima više adresnih linija nego što ISA koristi
    - P4: 36 linija, max  $|FAP| = 64 \text{ GB}$
    - pojedini korisnici transparentno koriste LAP od 4GB
    - ako korisnik želi više od 4GB, mora moliti OS za pomoć

$|LAP| > |FAP|$ : najčešći slučaj (sve vrste računala)

- poželjno zbog mogućnosti lakog proširenja  
(razlog za 64-bitne procesore)

# Adresno preslikavanje

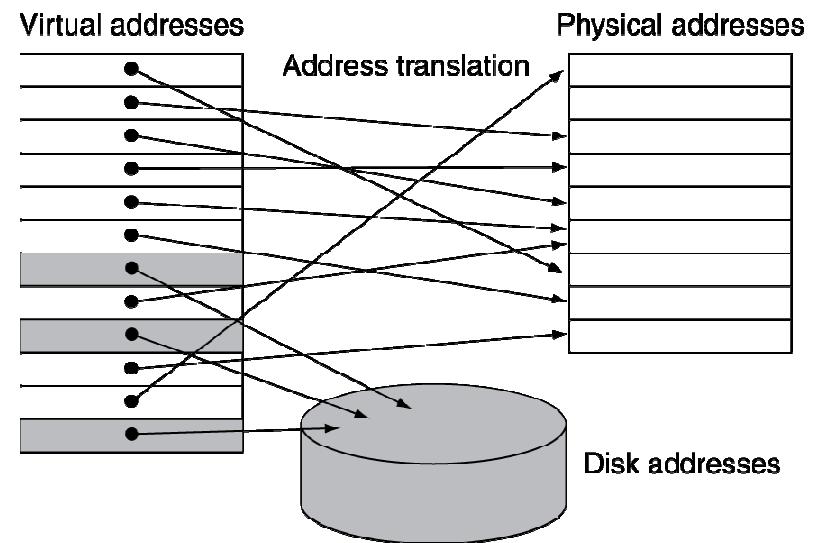
- ideja: svaki proces radi u **privatnom** virtualnom LAP-u
- OS odlučuje kako se mapira koji LAP (nema fragmentacije, zaštita, ušteda)
- adresno preslikavanje ključni detalj priče, obavlja se uz pomoć **sklopovlja**



# Svojstva funkcije preslikavanja

Za najčešći slučaj  $LAP \gg FAP$  funkcija preslikavanja  $f$  je:

$$f: LAP \rightarrow FAP \cup \emptyset$$



[Patterson08]

Za svaki  $a \in LAP$ , funkcija  $f$  vraća:

$f(a) = a'$       ako se podatak s virtualnom adresom  $a$  nalazi na adresi  $a'$  u fizičkoj memoriji ( $a' \in FAP$ )

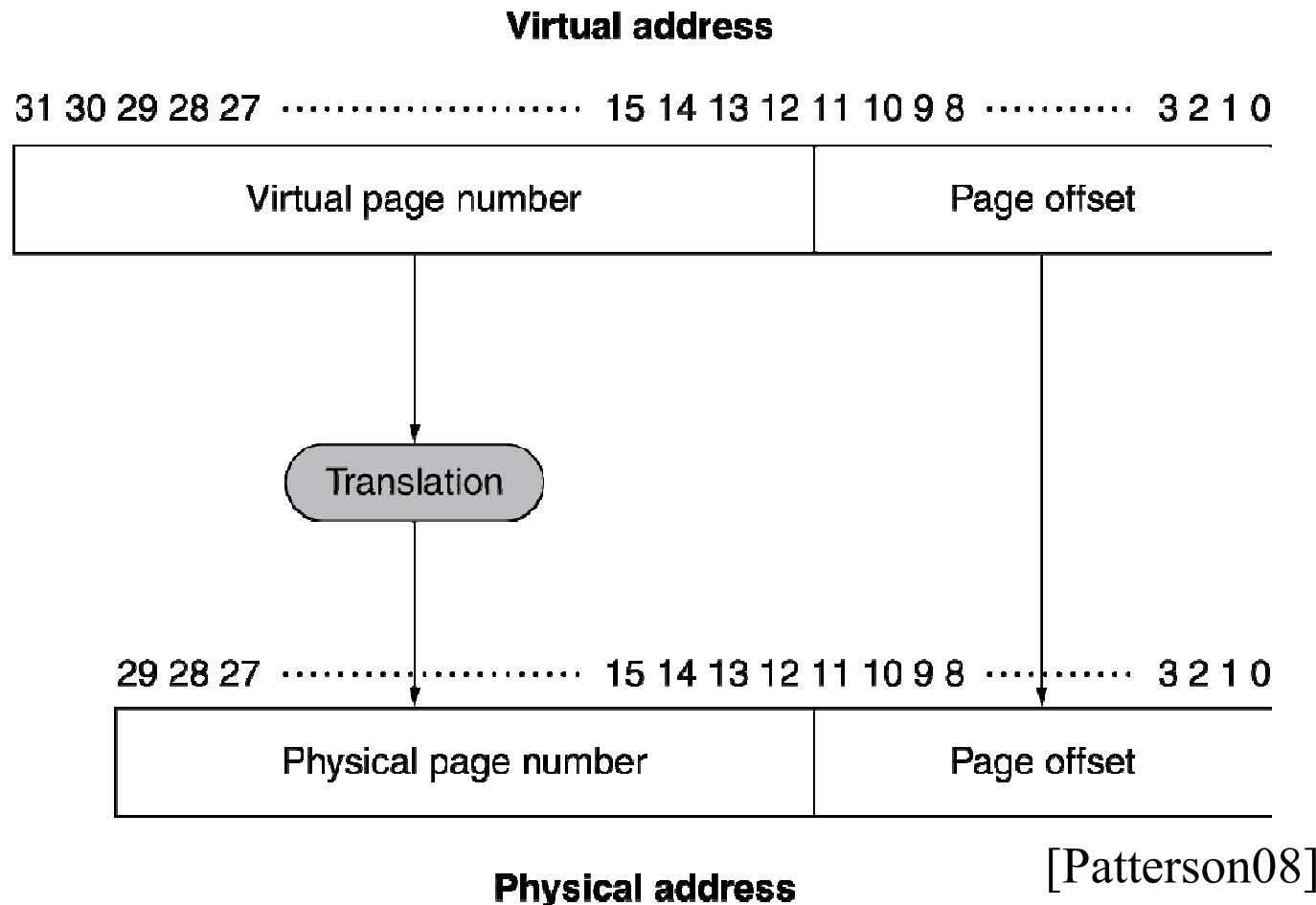
$f(a) = \emptyset$       označava **promašaj** virtualne memorije

# VIRTUALNI MEMORIJSKI SUSTAV

1. Memorijska hijerarhija
2. Fizički i logički adresni prostor
3. **Straničenje**
4. Višerazinsko straničenje, translacijski spremnik
5. Segmentacija
6. Detalji izvedbe

## Straničenje (engl. Paging)

- najčešće korištena implementacija VM, analogna cachevima:
  - preslikavaju se memorijski **blokovi** fiksne veličine
  - blokove nazivamo stranicama (tipična veličina stranice je 4KB)



## Straničenje se temelji na straničnoj tablici (ST):

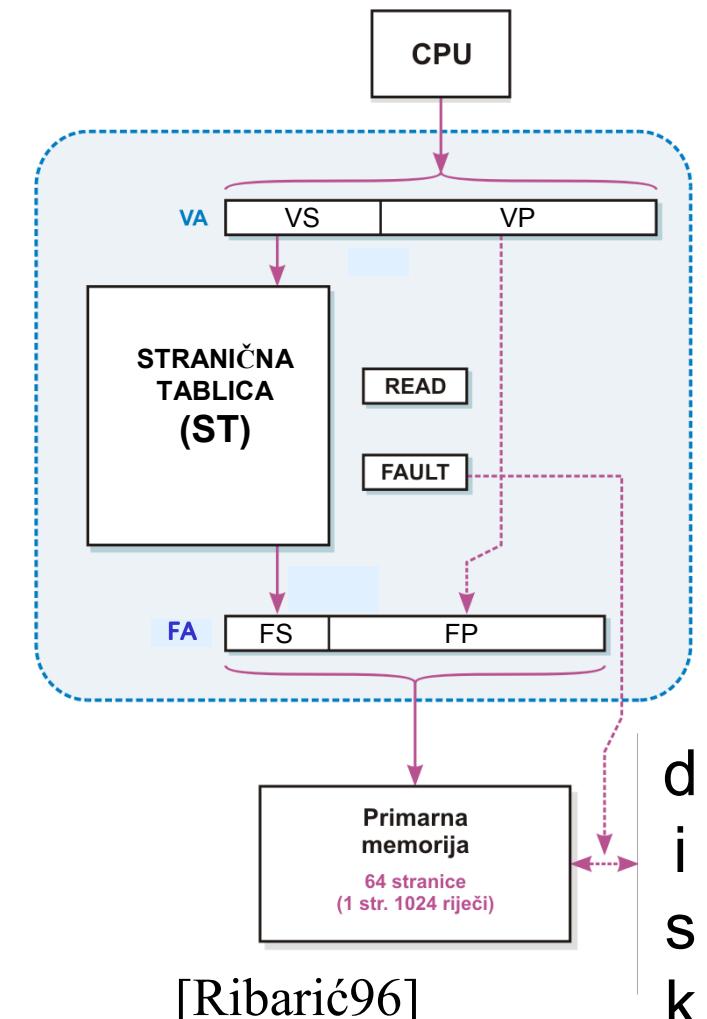
- virtualnu adresu (VA) dijelimo na dva dijela (kao kod cacheva)



- **virtualna stranica (VS)** adresira ST
- **virtualni pomak (VP)** adresira riječ stranice

## Određivanje fizičke adrese (FA):

- **okvir:** FS = ST (VS)
- **fizički pomak:** FP = VP!
- analogno priručnoj memoriji!



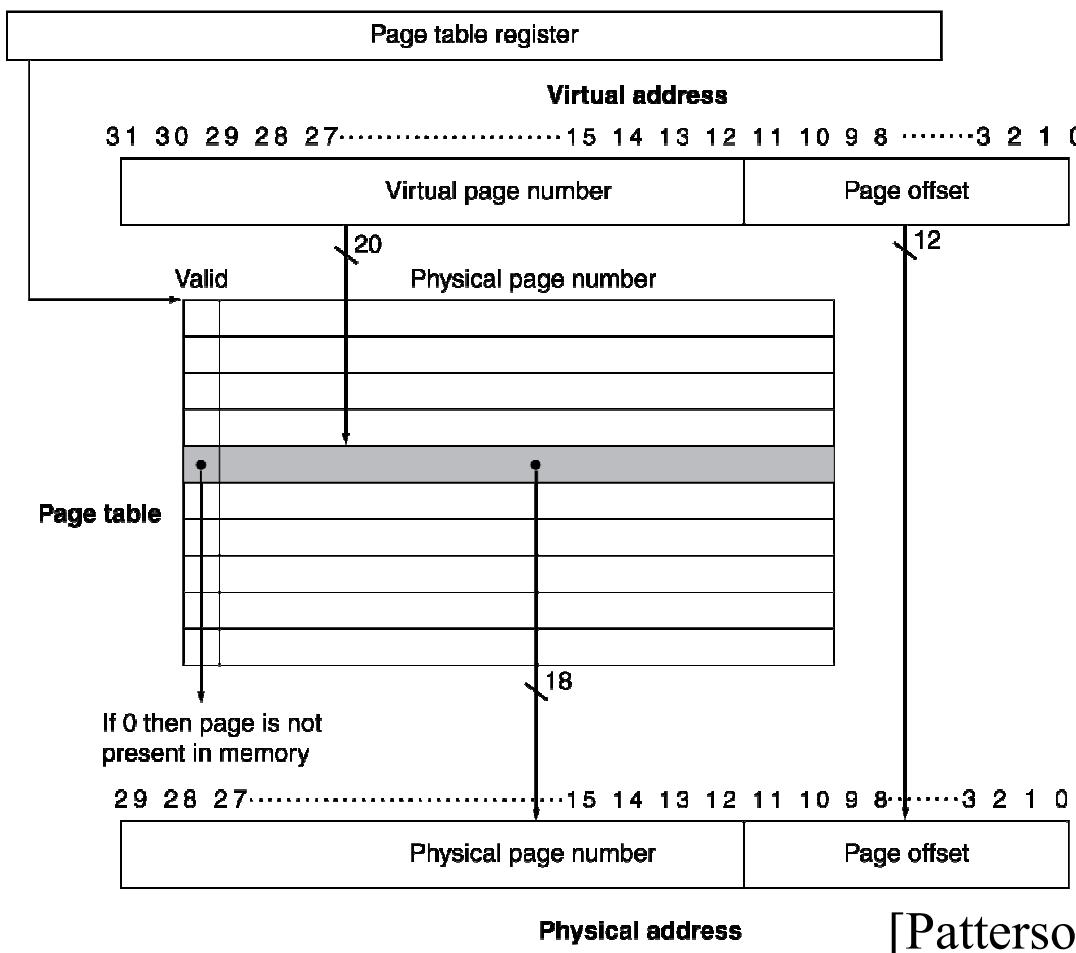
## Servisiranje promašaja VM (page fault):

- stranica mora biti učitana s diska ( $\sim 1e6 \Delta T$ )
- za to se brine operacijski sustav
- sofisticirane tehnike minimiziranja promašaja

[Ribarić96]

## ST se smješta u radnu memoriju

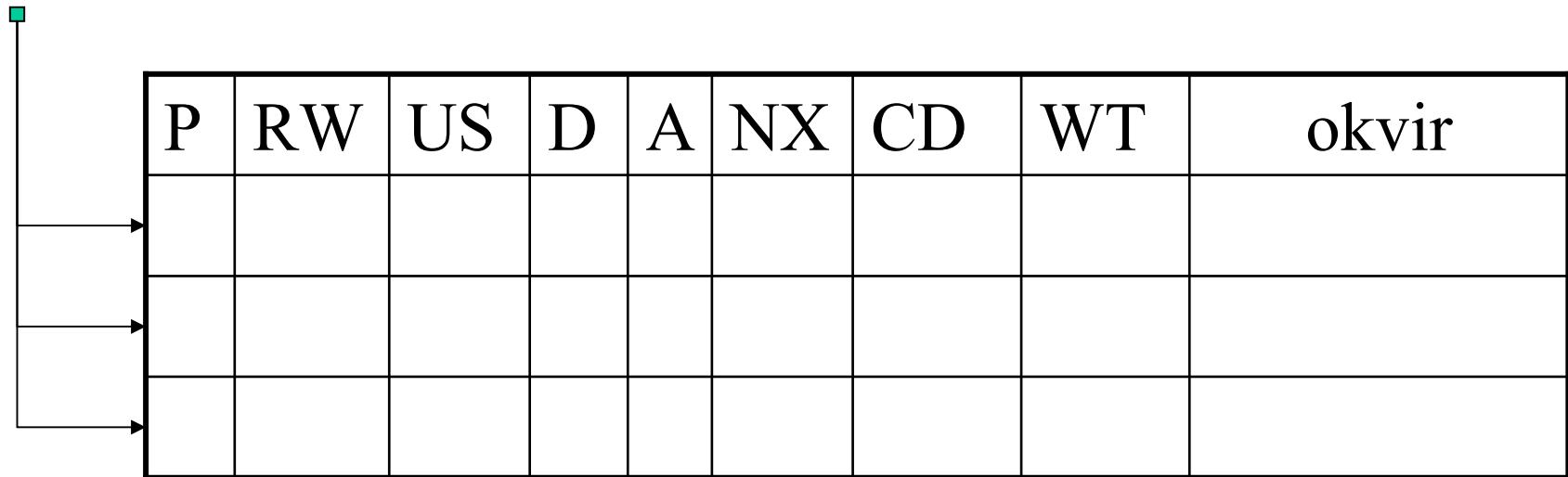
- element procesa, kao i registri (PC, ...)
- organizacija i održavanje u domeni OS-a
- pokazivač na ST procesa u posebnom registru



## Još o straničenju

- Najčešće korištena implementacija VM
  - stranice su jedinica transfera prema sekundarnoj memoriji
  - **prednosti**: transparentno preslikavanje, nema **vanjske** fragmentacije
  - **nedostatci**: **unutrašnja** fragmentacija, složena implementacija
- Elementi ST (**stranični opisnici**) pohranjuju dodatne informacije o stranici:
  - bitovi kontrole pristupa (zaštite):
    - RO (read only)
    - NX (no execute)
    - S (supervisor only)
  - ostale servisne informacije:
    - D (dirty), A (accessed),
    - WT (write through), CD (cache disable), ...
  - omogućava se sofisticirano upravljanje memorijom (OS)

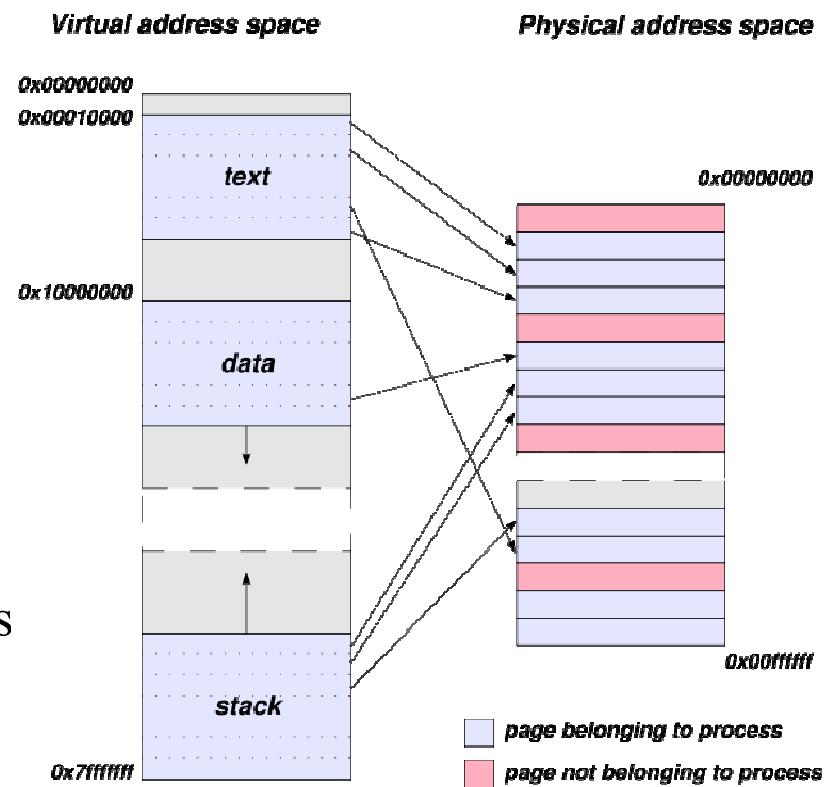
# Elementi ST: stranični opisnici (AMD Opteron)



- P (presence): 0  $\Rightarrow$  podatak je na disku
- RW (read/write): samo čitanje ili čitanje i pisanje
- US (user/supervisor): aplikacije/OS
- D (dirty): sadržaj je mijenjan
- A (accessed): da li je proces pristupao stranici?
- NX (no-execute): zabrana izvršavanja
- CD (cache-disable): zabrana cacheiranja
- WT (write-through): trenutni upis u fizičku memoriju

# Svojstva straničenja

- linearne logičke adrese:
  - straničenje rješava fragmentaciju
  - proces može dinamički **rasti**
    - npr, treba više prostora na stogu:
    - adresiramo nemapirani dio LAP-a
    - dolazi do iznimke (page fault)
    - OS **umeće** nove stranice u LAP
  - programi mogu imati apsolutne adres
- zaštita i privatnost:
  - svaki proces ima **privatnu ST**
  - proces ne može negativno utjecati na druge procese ili jezgru
- povećanje dostupnog prostora
  - ograničenje: veličina rezerviranog prostora na disku (swap file)
- mogućnost **dijeljenja** fizičkih stranica (biblioteke!)



[Wikipedia]

# VM straničenjem: **problemi**

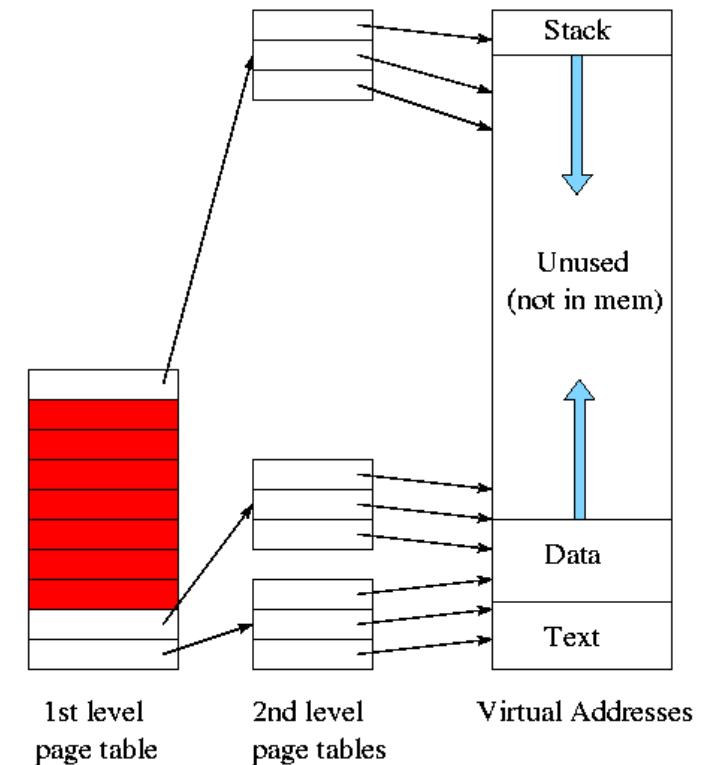
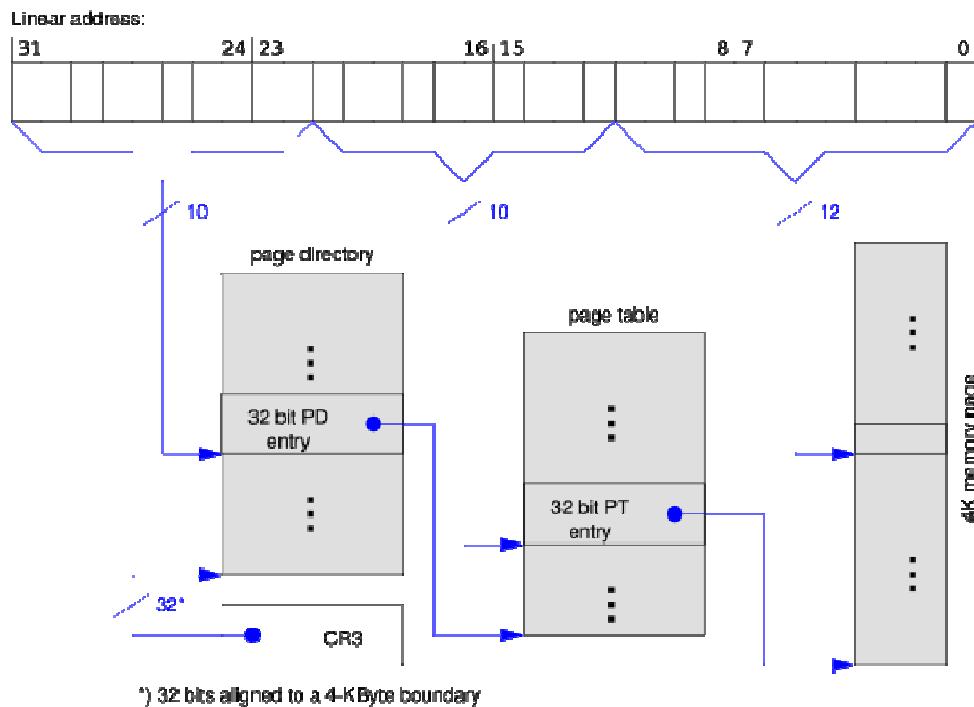
1. značajan prostor potreban za linearu ST
  - 32bit LAP, 4KB stranice  $\Rightarrow$  20b oznake  
 $\Rightarrow$  4MB za **svaki proces** (pa i onaj najmanji)!
  - a što je s računalima koja imaju 64-bitni LAP???
  - rješava se **višerazinskim ST**
2. svaki logički pristup  $\Rightarrow$  (barem) dva fizička pristupa
  - pristup ST (više njih?) + pristup podatku
  - a što je s pristupnim bitovima (D,A,R/W, ...)?
  - rješava se **cacheiranjem opisnika ST** (translacijski spremnik, TLB)
3. što ako **radni skup** (popularne stranice)  $>$ FAP?
  - u najgorem slučaju, svaki podatak dohvaćamo s diska
  - gubimo vrijeme u žongliraju stranicama - thrashing (mlaćenje)
  - koncept VM **nije prikladan** u takvim slučajevima

# VIRTUALNI MEMORIJSKI SUSTAV

1. Memorijska hijerarhija
2. Fizički i logički adresni prostor
3. Straničenje
- 4. Višerazinsko straničenje, translacijski spremnik**
5. Segmentacija
6. Detalji izvedbe

## Višerazinske stranične tablice

- **Problem** linearne ST: mali procesi plaćaju punu cijenu preslikavanja!
- **Višerazinska ST:** manja cijena nekorištenih dijelova LAP-a (prilagodljivo tabeliranje)
- Dvorazinsko straničenje x86 (nakon 80386):

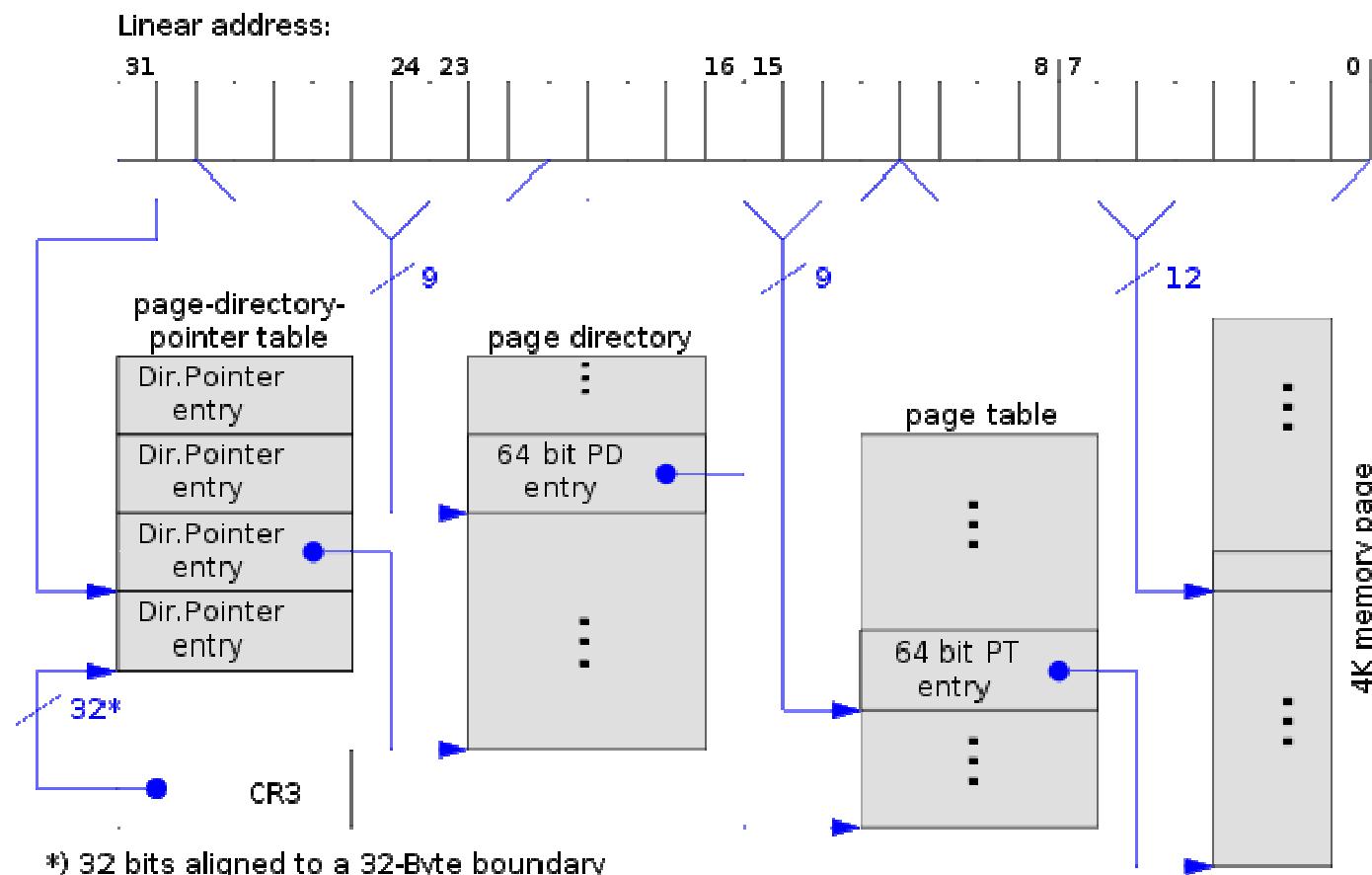


[Gottlieb00]

[Wikipedia]

# Trorazinsko straničenje x86 (Pentium II →)

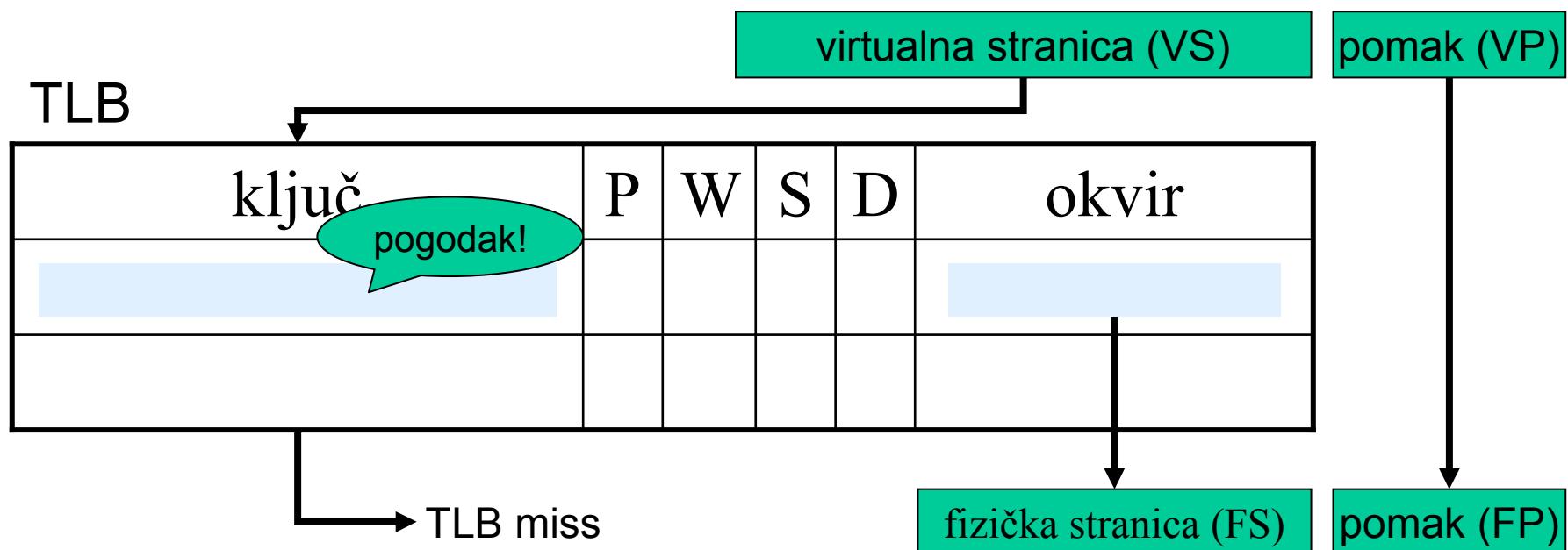
- ekstenzija PAE: physical address extension
- $|FAP|=2^{36} B$  (pojedinačni procesi koriste  $|LAP|=2^{32} B$ )



[Wikipedia]

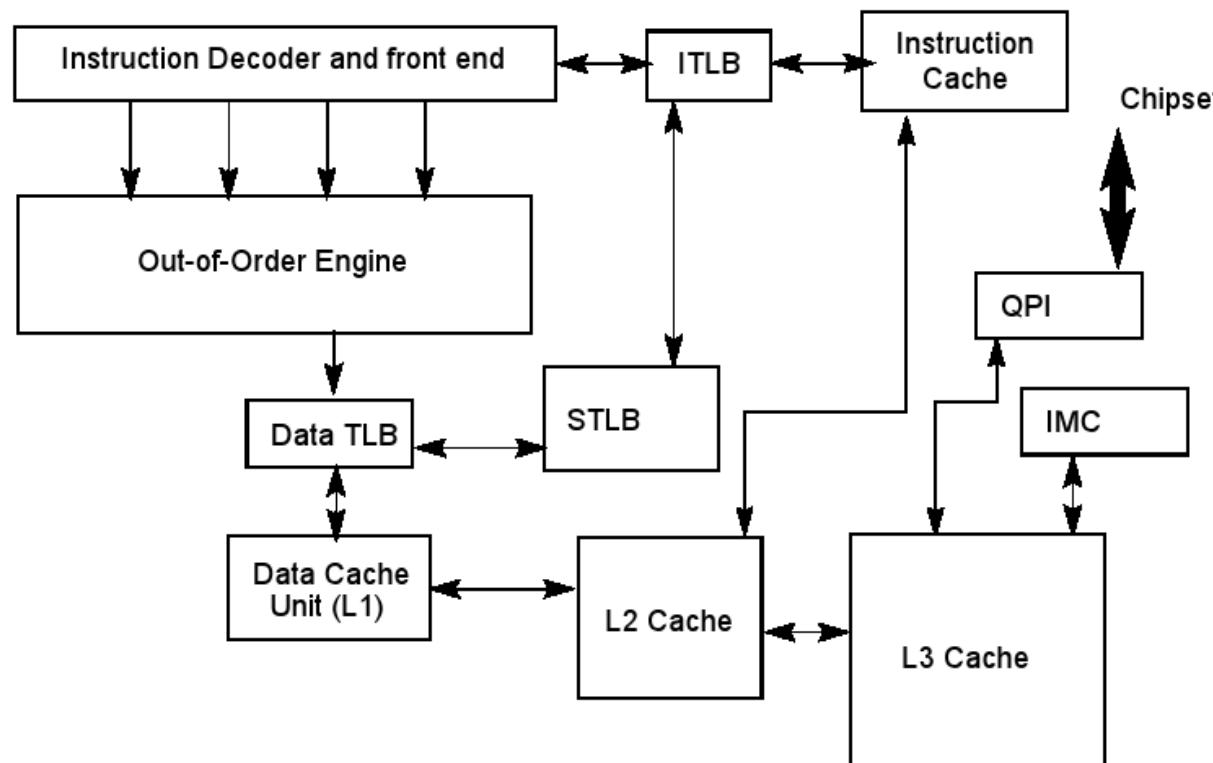
# Korištenje translacijskog spremnika (TLB)

- **Problem:** više fizičkih pristupa memoriji  $\forall$  preslikavanje
- **Ideja:** cacheirati posljednjih nekoliko stotina preslikavanja
  - TLB: **asocira** virtualnu stranicu s odgovarajućim SO (kao kod PM!)
  - u slučaju pogotka, gotovi smo u jednom ciklusu
  - u slučaju promašaja, šetnja straničnom tablicom te upis opisnika u TLB
  - punjenje TLB-a može biti programsko (OS) ili sklopovsko (MMU)



## Odnos TLB-a i priručne memorije

- Najčišće kad je (barem koncepcualno) TLB ispred cachea
  - **dobro**: u cacheu ne mogu biti dvije LA koje se odnose na istu FA!
  - **problem**: tada latencija TLB-a produžava latenciju pristupa cacheu
  - to se rješava na razne komplikirane načine
    - ti načini su izvan dosega ovog kolegija :-)



[intel.com]

## TLB, izvedbeni detalji

- veličine variraju (8 - 4096 zapisa), kao i broj razina (1 ili 2)
- kao i kod PM, kombinira se nepotpuno indeksiranje i asociranje
- manji TLB-ovi potpuno asocijativni, veći skupno asocijativni
  - ST koristi potpuno indeksiranje:
    - rijetko se proziva, višerazinska struktura štedi prostor
    - želimo iskoristiti sve stranične okvire (tj, fizičku memoriju)
- postotak pogotka vrlo visok, promašaji  $\sim 0.1\%$   
(stranica veća od linije, 4096B vs. 64B)
- cijena promašaja niska, pogotovo ako je ST u cacheu

## Zadano:

- virtualna adresa:  $w(VA)=40$  b
- fizička adresa:  $w(FA)=36$  b
- veličina stranice: 16 kB

Odrediti strukturu VA i FA:

$$w(VS), w(FS), w(VP), w(FP)=?$$

- $w(VP) = w(FP) = 14$  b
- $w(VS) = 26$  b
- $w(FS) = 22$  b

## Zadano:

- virtualna adresa:  $w(VA)=40$  b
- fizička adresa:  $w(FA)=36$  b
- veličina stranice: 16 kB
- TLB: 512 zapisa,  $2\times$  asocijativan, 4 bita opisa stranice

Odrediti strukturu i ukupan broj bitova zapisa u TLB-u  
(oznaka, bitovi opisa, fizička stranica):  $w(TLBo) + 4 + w(FS) = ?$

## Rješenje:

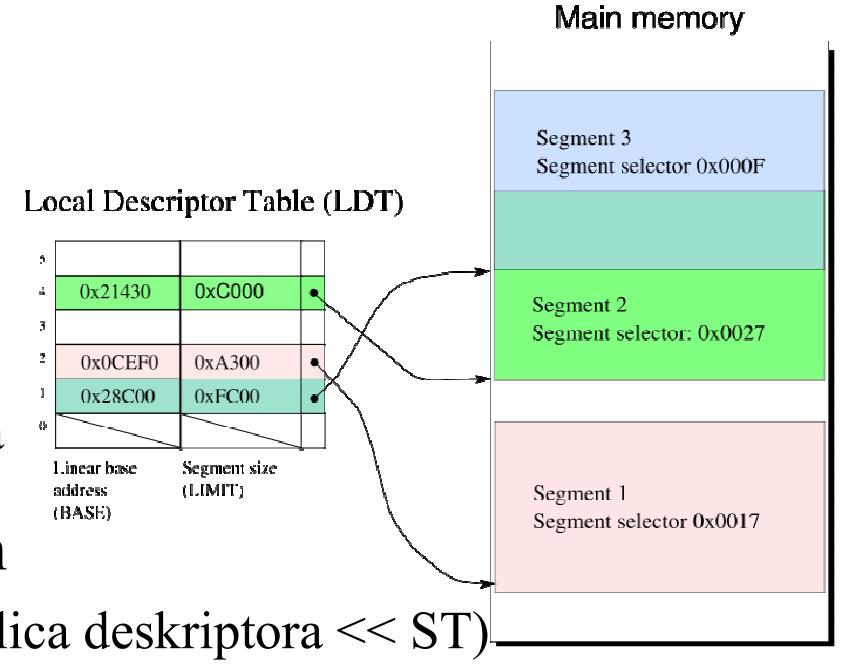
- VS → TLBO || TLBI (kao i kod PM!)
- $w(TLBindeks) = 8$  b                ( $= \log_2(512/2)$ )
- $w(TLBoznaka) = 18$  b                ( $= 40 - 14 - 8$ )
- $w(TLBzapis) = 44$  b                ( $= 18_{TLBozn} + 4 + 22_{FS}$ )

# VIRTUALNI MEMORIJSKI SUSTAV

1. Memorijska hijerarhija
2. Fizički i logički adresni prostor
3. Straničenje
4. Višerazinsko straničenje, translacijski spremnik
- 5. Segmentacija**
6. Detalji izvedbe

# Segmentacija

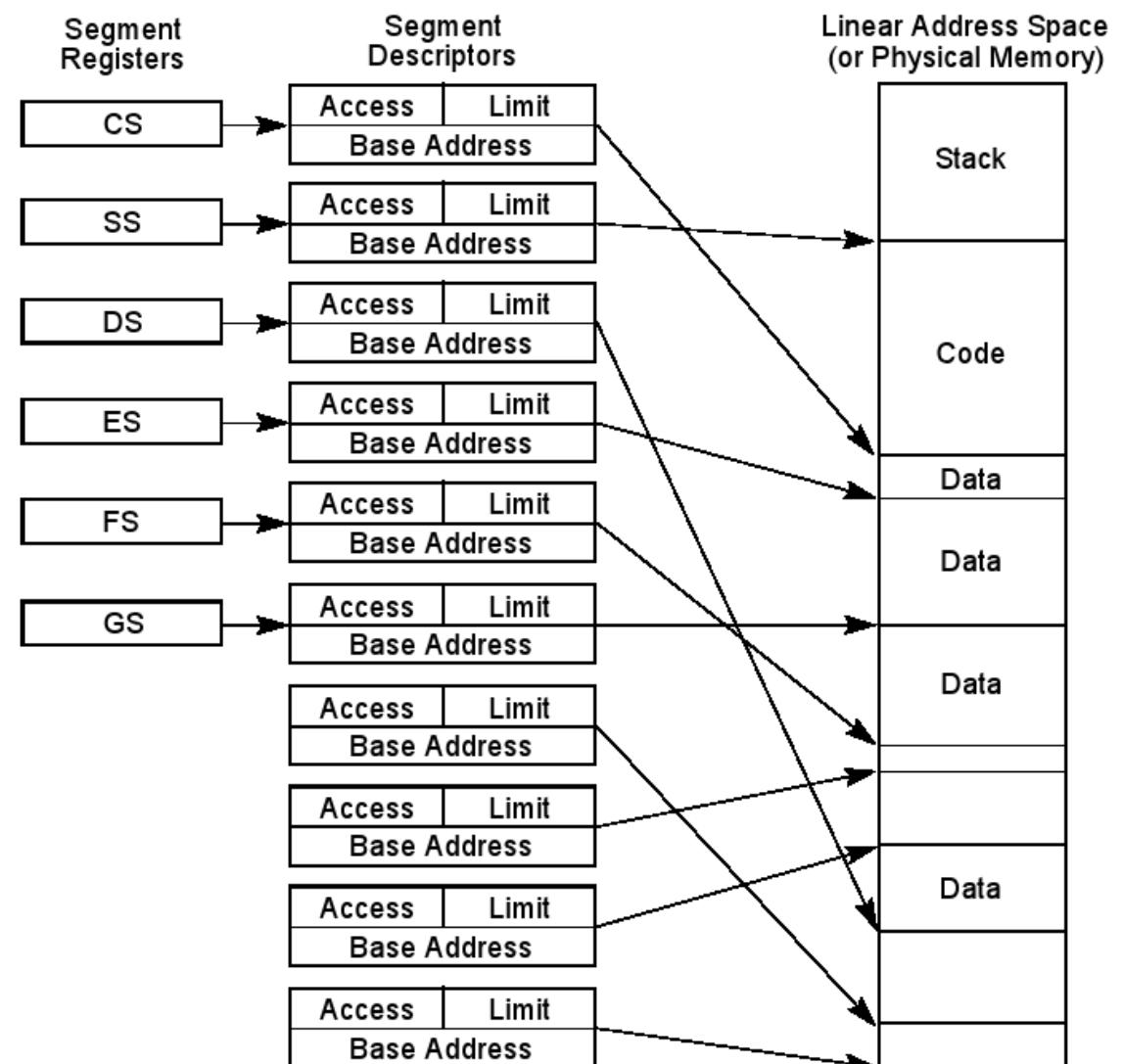
- Razlike u odnosu na straničenje:
  - blokovi su veći i promjenljive duljine
  - LAP procesa čini nekoliko segmenata
- **prednost:** preslikavanje zbrajanjem
  - jednostavno ( $FA = S + LA$ ), jeftino (tablica deskriptora  $\ll ST$ )
  - (početak i duljina segmenta, prava pristupa)
- **nedostatak:** upravljanje memorijom na pregruboj razini
  - vanjska fragmentacija: neiskorišteni prostor između segmenata (teško naći kontinuirani logički interval u kojem preslikati novi segment)
  - netransparentan pristup kad:
    - iscrpimo mogućnosti rasta tekućeg segmenta
    - pristupamo dijeljenim segmentima (potreban poseban tip pokazivača)
- arhitektura x86 podržava i segmentaciju i straničenje (moderni OS-ovi koriste samo straničenje)



[Wikipedia]

# Segmentacija x86

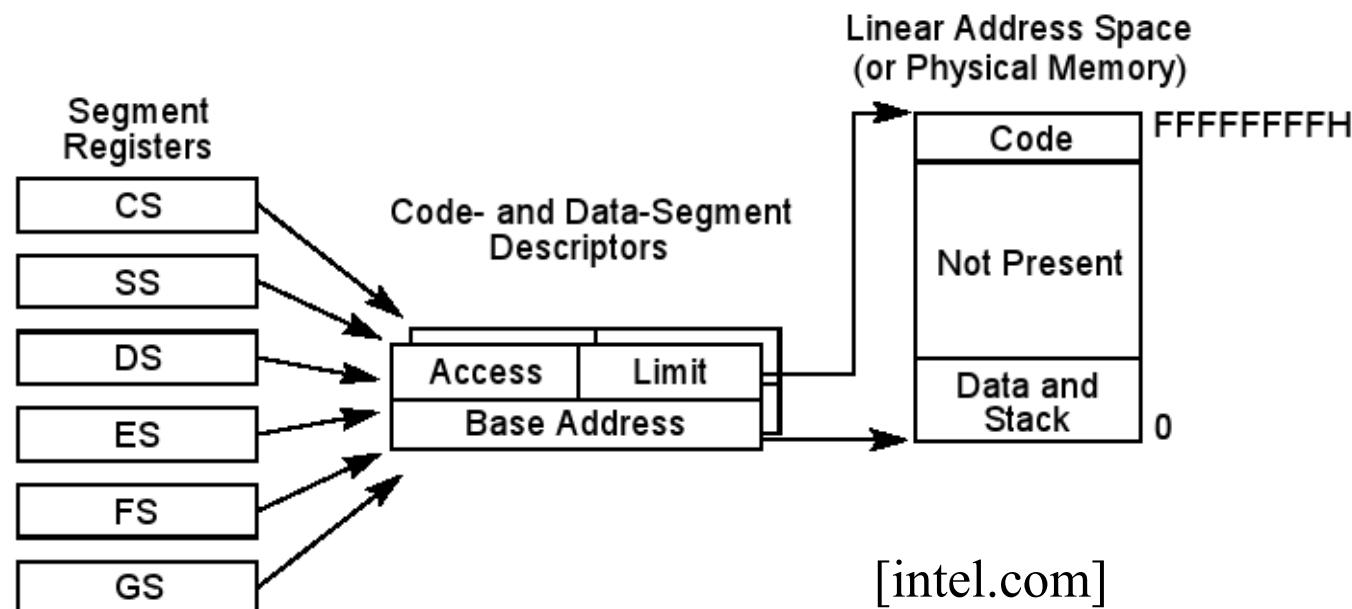
- pristupom memoriji upravlja 6 segmentnih registara
- svaki segmentni opisnik sadrži bitove zaštite i veličinu
- nedostatak: pokazivač na podatke izvan segmenta ima 48 bitova
- nedostatak: vanjska fragmentacija
- nedostatak: što ako je segment nedovoljno velik?



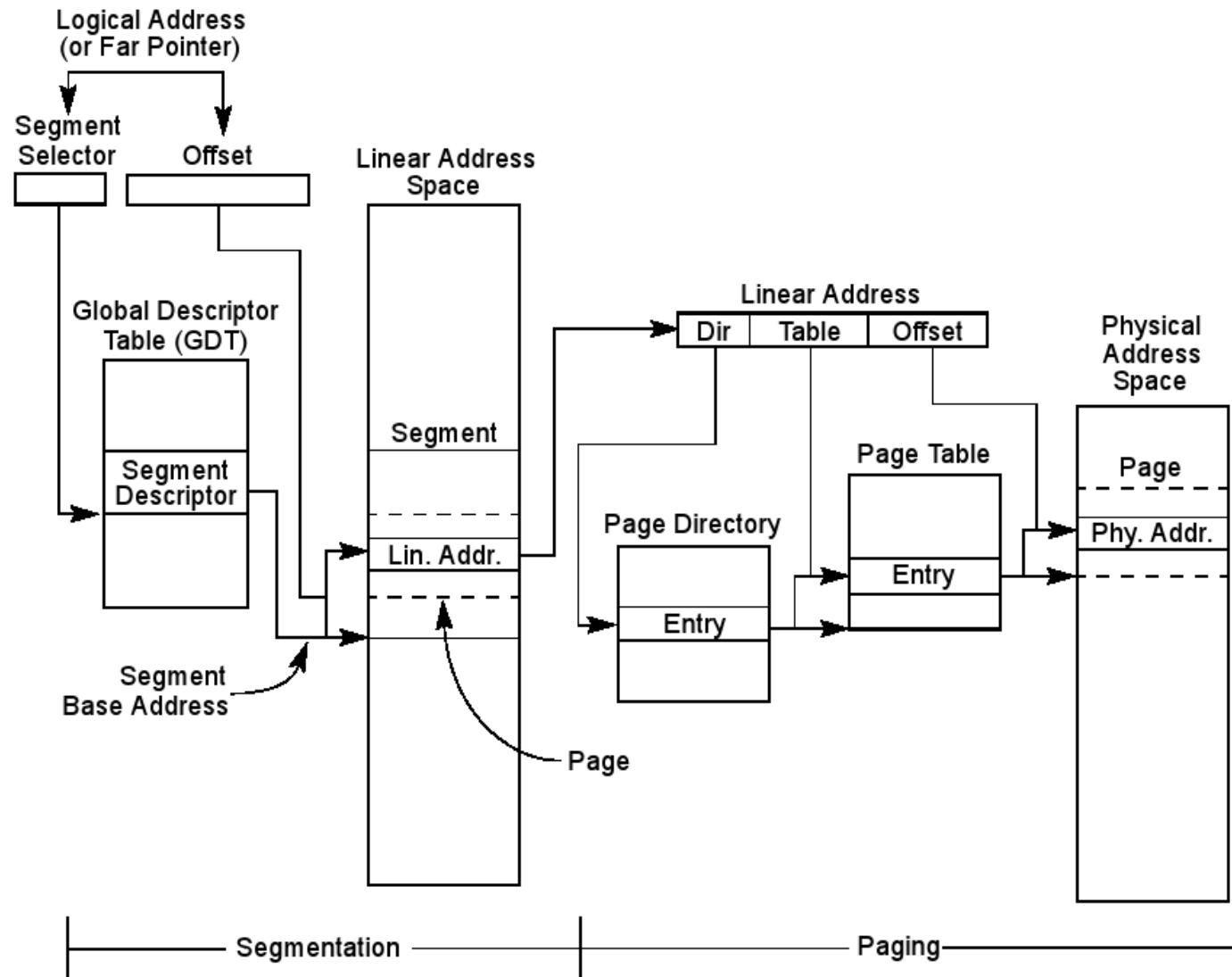
[intel.com]

## Segmentacija x86 (2)

- moderni OS-ovi konfiguiraju minimalno korištenje segmentacije
- Intel taj način naziva “basic flat model” (vidi sliku)
- preslikavanje i zaštita se prepušta straničenju
- segmentacija danas postoji prvenstveno zbog kompatibilnosti
- AMD Opteron u 64-bitnom načinu rada nema segmentaciju



# VM x86: kombinacija segmentacije i straničenja



[intel.com]

# VIRTUALNI MEMORIJSKI SUSTAV

1. Memorijska hijerarhija
2. Fizički i logički adresni prostor
3. Straničenje
4. Višerazinsko straničenje, translacijski spremnik
5. Segmentacija
6. **Detalji izvedbe**

# Obrada promašaja stranice

Promašaj stranice generira iznimku koju servisira OS:

1. locirati SO na temelju adrese koja je uzrokovala promašaj
  - šetnja višerazinskom ST (ona bi trebala biti u RAM-u)
2. ako SO nije valjan (tj. referenciramo novu fizičku stranicu):
  - ili mapirati novu stranicu (npr. pri širenju stoga)
  - ili likvidirati program zbog ilegalnog pristupa
3. ako je fizička memorija popunjena, odabrat stranicu koju ćeemo izbaciti (swap-out, evict)
  - ako smo u stranicu pisali (bit D), upisujemo je na disk (**dugo čekanje**)
4. ako je SO valjan: učitati stranicu s diska (**dugo čekanje**)
5. ažurirati ST, premjestiti proces u red aktivnih procesa
  - nastavak izvođenja od instrukcije koja je generirala promašaj

## Zamjena stranice, upis na disk

- najčešće potpuno asocijativno preslikavanje + LRU
  - potpuna asocijativnost izvediva zbog ogromne ST u RAM-u
  - izvedba LRU se temelji na servisnom bitu pristupa (A):
    - postavlja se na 1 kod svakog pristupa (TLB)
    - operacijski sustav periodički poništava sve bitove pristupa
    - stranica s  $A = 0$  nije korištena u bliskoj prošlosti
- tehnike upisa na disk:
  - promptno upisivanje (write through) uglavnom nepraktično
  - tipično, stranica se upisuje na disk tek nakon zamjene (write back)
  - servisni bit D (dirty) postavlja se u SO nakon svakog upisa

## Performansa memorijskog sustava

- možemo dobiti promašaj u: TLB-u, SO-u, cacheu, DRAM-u
- neki od mogućih ishoda:
  - pogodak TLB-a i cachea (zaobilazimo radnu memoriju): 2 ns
  - pogodak TLB-a, promašaj cachea, pogodak DRAM: 50 ns
  - promašaj TLB-a, SO u cacheu, pogodak cachea: 10 ns
  - promašaj TLB-a, SO u DRAM-u, promašaj cachea, pogodak DRAM: 100 ns
  - promašaj TLB-a, SO u DRAM-u, promašaj cachea, promašaj DRAM: 10 ms  
(tijekom tih 10ms OS pokušava dodijeliti CPU nekom drugom procesu)
- širok spektar mogućnosti i to bez razmatranja PM L2, PM L3, TLB L2!
- vrijeme pojedinog pristupa memoriji podataka vrlo teško predvidjeti  
(prisjetimo se, u prosjeku svaka treća instrukcija pristupa memoriji)
  - pristupi instrukcijskoj memoriji znatno pravilniji!
- u prisustvu nepredvidivih zastoja u vezi s memorijom (i predviđanjem grananja) prednost imaju procesori s dinamičkim raspoređivanjem!

## Usporedba virtualne i priručne memorije [Hennesy07]

| parametar             | PM (L1)                           | VM                                |
|-----------------------|-----------------------------------|-----------------------------------|
| veličina bloka        | 16 - 128 B                        | 4 - 64 kB                         |
| vrijeme pogotka       | 1 - 3 $\Delta T$                  | 100 - 200 $\Delta T$              |
| vrijeme promašaja     | 8 - 200 $\Delta T$                | $10^6$ - $10^7 \Delta T$          |
| učestalost promašaja  | $10^{-3}$ - $10^{-1}$             | $10^{-7}$ - $10^{-5}$             |
| adresno preslikavanje | 25 - 45 b $\rightarrow$ 14 - 20 b | 32 - 64 b $\rightarrow$ 25 - 45 b |

## Virtualna memorija, prednosti:

1. transparentno adresno preslikavanje
  - ogromni kontinuirani LAP, fragmentirani FAP
  - nekorišteni dijelovi programa ostaju na disku
2. transparentna zaštita procesa i jezgre OS
  - ograničena neželjena interakcija među procesima
  - upravljanje pristupom memoriji (RO, NX, S, ...)
3. transparentno rukovanje dijeljenom memorijom
  - dijeljene biblioteke (libc, libm, ...)
  - usporedna obrada više procesa

## TLB ključna tehnika za implementaciju straničenja:

- svaki pristup memoriji zahtijeva adresno preslikavanje
- TLB omogućava dovoljno brzo preslikavanje u najčešćem slučaju

Prof.dr.sc. Slobodan Ribarić

## **Višeprocesorski sustavi, višejezgreni i grafički procesori**

**(Grada računala, Arhitektura i organizacija računarskih sustava, str. 473-532)**

1. Oblici i razine paralelizma
2. Paralelne arhitekture
3. Višeprocesorski SIMD sustavi
4. Vektorski procesori
5. Multiprocesorski sustavi – višeprocesorski MIMD sustavi
6. Koherencija priručne memorije u multiprocesorskom sustavu
7. Sinkronizacija procesa i dretvi
8. Višejezgreni procesori

## 5. Multiprocesorski sustavi – višeprocesorski MIMD sustavi

MIMD arhitektura obilježena je višestrukim instrukcijskim tokom i višestrukim tokom podataka. Svaki od procesora pribavlja i izvršava svoje vlastite instrukcije na svojim podacima.

Višeprocesorski MIMD sustavi ili *multiprocesorski sustavi* iskorištavaju *paralelizam na razini procesa i dretvi*.

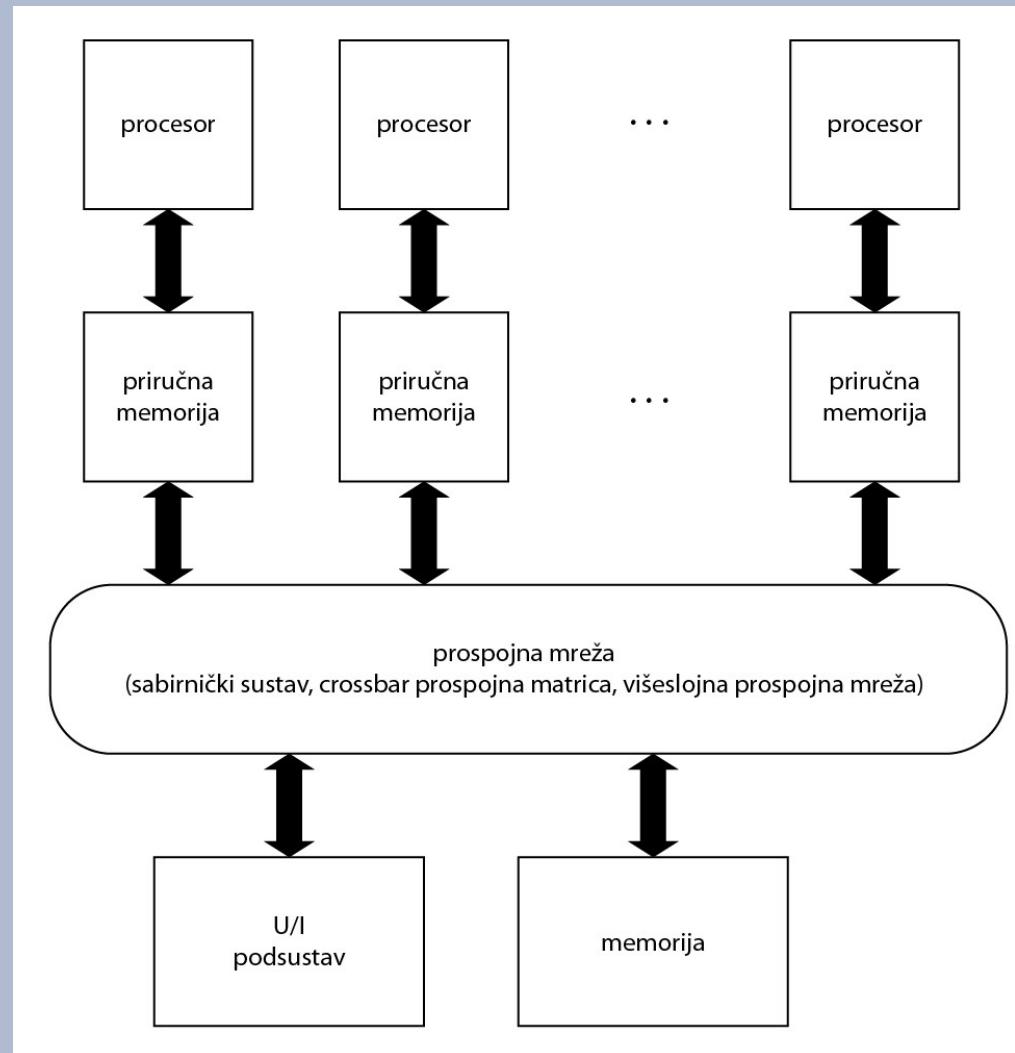
- U multiprocesorskom sustavu svaki procesor može izvršavati njemu dodijeljen *proces*.
- Svaki od procesa može imati više dretvi tako da se izvođenje jednog procesa s većim brojem dretvi može povjeriti većem broju procesora. **Višedretvena arhitektura** temeljena na MIMD-u, u načelu, dopušta istodobno izvođenje većeg broja procesa s izdvojenim adresnim prostorima i izvođenje više dretvi koje dijele adresni prostor.

Osnovna značajka multiprocesorskog sustava jest veći broj procesora približno jednakih (vrlo često identičnih) obilježja koji na izvjestan način dijele zajednički memorijski prostor.

Ovisno o broju procesora i načinu organizacije memorijskog sustava multiprocesorski se sustavi mogu klasificirati u sljedeće skupine:

- **sustavi s uniformnim pristupom memoriji UMA** (engl. *Uniform Memory Access*),
- **sustavi s neuniformnim pristupom memoriji NUMA** (engl. *Nonuniform Memory Access*),
- **sustavi samo s priručnom memorijom COMA** (engl. *Cache-Only Memory Architecture*).

U *UMA modelu multiprocesorskog sustava* svi procesori dijele zajedničku fizičku memoriju i svi imaju jednako vrijeme pristupa svakoj od riječi u memoriji (uniformni, odnosno ujednačeni pristup memoriji). Svaki od procesora može imati i svoju vlastitu priručnu memoriju organiziranu u jednu ili više razina. Na sličan način kao što dijele memoriju, procesori dijele i resurse U/I podsustava.

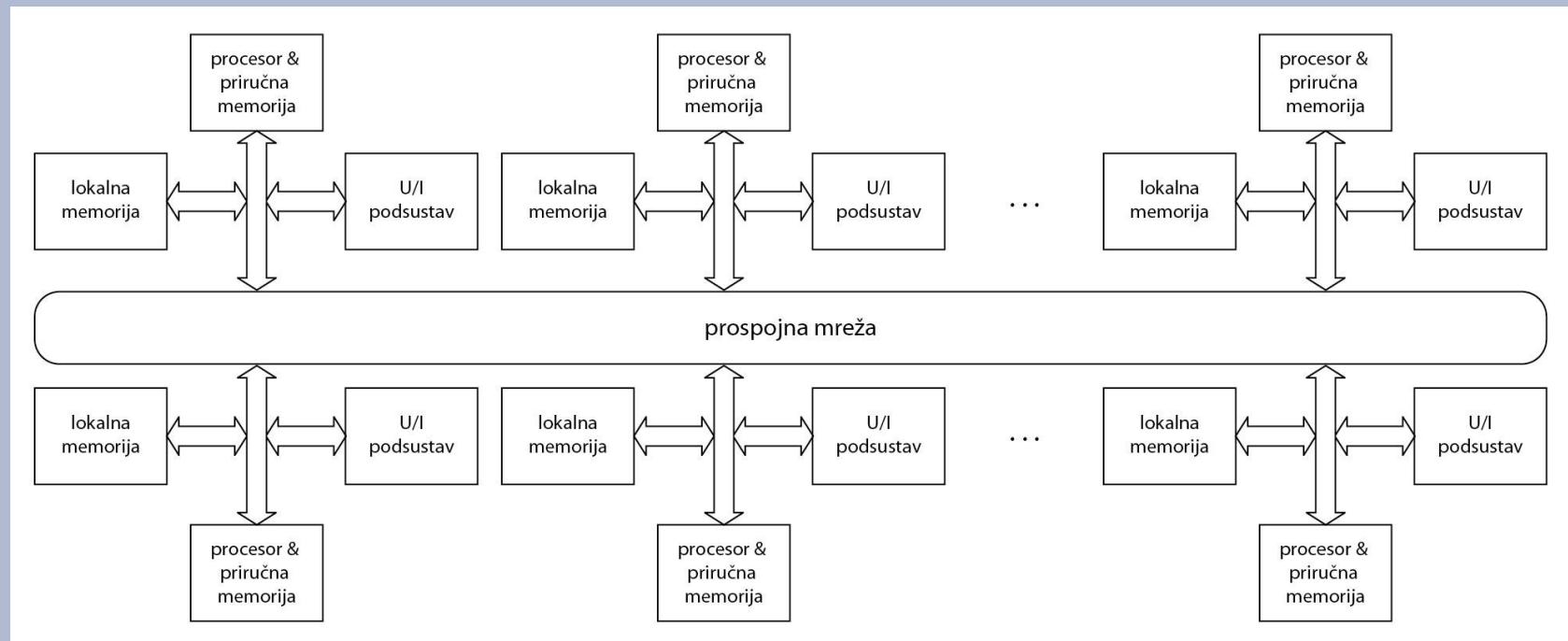


UMA model multiprocesorskog sustava

UMA model se još naziva i *multiprocesorski sustav sa središnjom dijeljenom memorijom* (engl. *centralized shared-memory*) ili *simetrični multiprocesorski sustav s dijeljenom memorijom SMP* (engl. *symmetric shared-memory multiprocessor*) – zato što memorija ima isti ("simetrični") odnos spram svih procesora.

- Komunikacija procesora sa zajedničkom memorijom i U/I podsustavom ostvaruje se prospojnom mrežom koja ovisno o zahtijevanoj propusnosti može biti ostvarena **sabirničkim** sustavom, **crossbar prospojnom matricom** ili **višerazinskom prospojnom mrežom** (npr. Omega).

*Multiprocesorski sustavi s neuniformnim pristupom memoriji NUMA* nazivaju se još i *sustavi s porazdijeljenom memorijom* (engl. *distributed-memory multiprocessor*) imaju umjesto centralizirane memorije, memoriju porazdijeljenu procesorima.



Zbirka svih lokalnih memorija oblikuje globalni adresni prostor kojem mogu pristupiti svi procesori u sustavu. Zbog takve organizacije memorije razlikujemo dvije vrste pristupa memoriji:

- **brzi pristup memoriji** (kraće vrijeme pristupa) kada procesor pristupa svojoj lokalnoj memoriji,
- **sporiji pristup** (dulje vrijeme pristupa) kada procesor pristupa "udaljenoj" memoriji koja je, zapravo, lokalna memorija nekog drugog procesora. Dulje vrijeme pristupa uzrokovano je dodatnim kašnjenjima jer se "udaljenoj" memoriji pristupa kroz prospojnu mrežu.

- Multiprocesorski sustavi oblikovani u skladu s modelom NUMA imaju veliki broj procesora, npr. nekoliko stotina ili tisuća, i zato se za toliki broj procesora teško može realizirati središnja memorija sa zahtijevanom, odnosno prihvatljivom propusnosti (engl. *memory bandwidth*).
- Svaki čvor u NUMA modelu sastoji od procesora, lokalne memorije, U/I podsustava i sučelja za pristup prospojnoj mreži. Ovisno o izvedbi NUMA modela, čvor može biti sastavljen od određenog broja procesora i lokalnih memorijskih modula tako da čini **procesorsku nakupinu** (engl. *processor cluster*).

## 8. Višejezgreni procesori

- **Višedretvenost** (engl. *multithreading*) podrazumijeva da se više dretvi izvodi u jednom procesoru tako da se izvođenje dretvi međusobno isprepliće, odnosno dretve se naizmjenično izvode u dodijeljenim funkcijskim jedinicama procesora uz nužno prospajanje konteksta sadržanog u tablici dretvi, i to nakon svake izmjene dretve.

Dva su glavna pristupa višedretvenosti:

- **finozrnata višedretvenost** (engl. *fine-grained multithreading*),
- **grubozrnata višedretvenost** (engl. *coarse-grained multithreading*).

*Finozrnata višedretvenost* podrazumijeva prospajanje dretvi nakon svake instrukcije.

- Procesori s takvom značajkom nazivaju se još i "bačvasti", odnosno *barrel* procesori

Te su instrukcije međusobno **nezavisne** jer pripadaju različitim dretvama tako da se protočna struktura djelotvorno iskorištava.

- **povećani broj promašaja priručne memorije zbog narušavanja lokalnosti programa.** Obično se izvođenje dretvi u tom slučaju temelji na kružnom prioritetu (engl. *round-robin*) uz "preskakanje" dretvi koje su u stanju zastoja.

Višedretveni procesori imaju sklopovski podržano brzo prospajanje konteksta dretvi (tzv. *hardware based fast context switching*) koje se temelji na tome da svaka dretva ima dodijeljene fizičke registre za pohranu konteksta dretvi

Primjer:

Višedretveni procesor Tera podržava 128 dretvi, pri čemu je svakoj dretvi dodijeljen 41 64-bitni registar u procesoru. Jezgre, osim sklopovski podržanog brzog prospajanja konteksta, imaju i dinamičku izvedbu preimenovanja registara kojim se rješavaju hazardi WAW i WAR koji se nazivaju još i *lažne zavisnosti*

*Grubozrnata višedretvenost* predviđa prospajanje dretvi samo onda kada nastupa dulji zastoj u tekućoj dretvi (npr. promašaj u priručnoj memoriji). Za kraće zastoje, na primjer one izazvane u instrukcijskoj protočnoj strukturi uslijed hazarda, ne predviđa se izmjena dretvi te je to jedan od glavnih nedostataka grubozrnate višedretvenosti.

- Razlog da se ne koristi prospajanje dretvi kod zastoja izazvanih u protočnoj strukturi leži u procjeni **cijene i količine posla za pražnjenje protočne strukture da bi se u nju mogao uputiti instrukcijski tok druge dretve.**
- Izmjena dretvi i prospajanje njihova konteksta može se događati i pri svakoj *load* instrukciji (neovisno o tome je li se dogodio promašaj) ili nakon programskog odsječka (bloka instrukcija) koji pripadaju jednoj dretvi.

Grubozrnata i finozrnata višedretvenost može se kombinirati s paralelizmom na razini instrukcija ILP, ali i sa superskalarnosti (višestrukim protočnim strukturama u jednom procesoru). Tri su procesorske konfiguracije moguće u tom slučaju:

- superskalarnost s grubozrnatom višedretvenosti,
- superskalarnost s finozrnatom višedretvenosti,
- superskalarnost sa simultanom višedretvenosti.

Superskalarni procesori s gubozrnatom višedretvenosti izvode istodobno veći broj instrukcija koje pripadaju istoj dretvi sve do trenutka kada nastupi dulji zastoj, u tom se trenutku prospaja kontekst dretvi i nastavlja se s izvođenjem druge dretve.

- Superskalarni procesori s finozrnatom višedretvenosti isprepliću dretve i na taj način eliminiraju možebitne zastoje u izdavanju instrukcija. Budući da samo jedna dretva izdaje instrukcije tijekom periode signala vremenskog vođenja, još uvijek postoje ograničenja u paralelizmu na razini instrukcija. Višejezgreni procesori tvrtke Sun nazvani T1(Niagara 1) i T2 (Niagara 2) koriste finozrnatu višedretvenost.

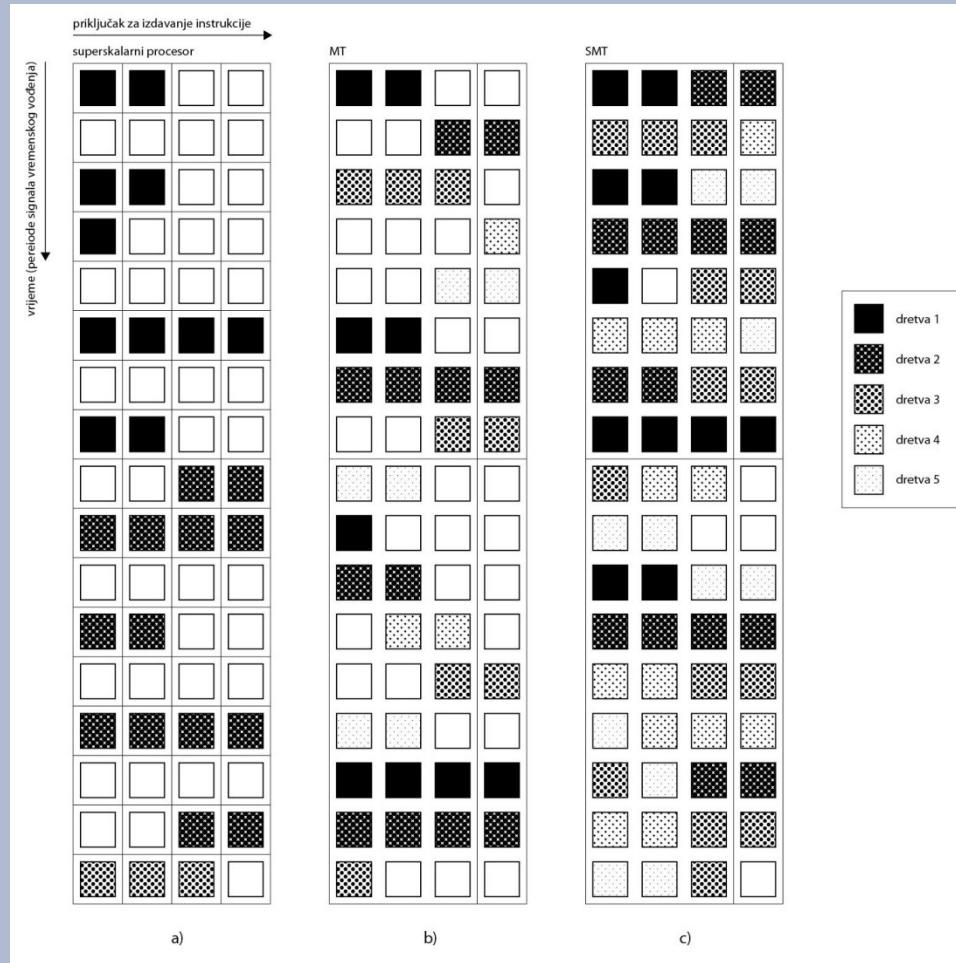
*Simultana višedretvenost SMT (engl. simultaneous multithreading)* zasniva se na činjenici da suvremeni (superskalarni) procesori imaju paralelizam na razini funkcijskih jedinica veći od onog koji jedna dretva može iskoristiti.

- dinamičkim raspoređivanjem izdaje *istodobno* više instrukcija iz *nezavisnih* dretvi u istoj periodi signala vremenskog vođenja.

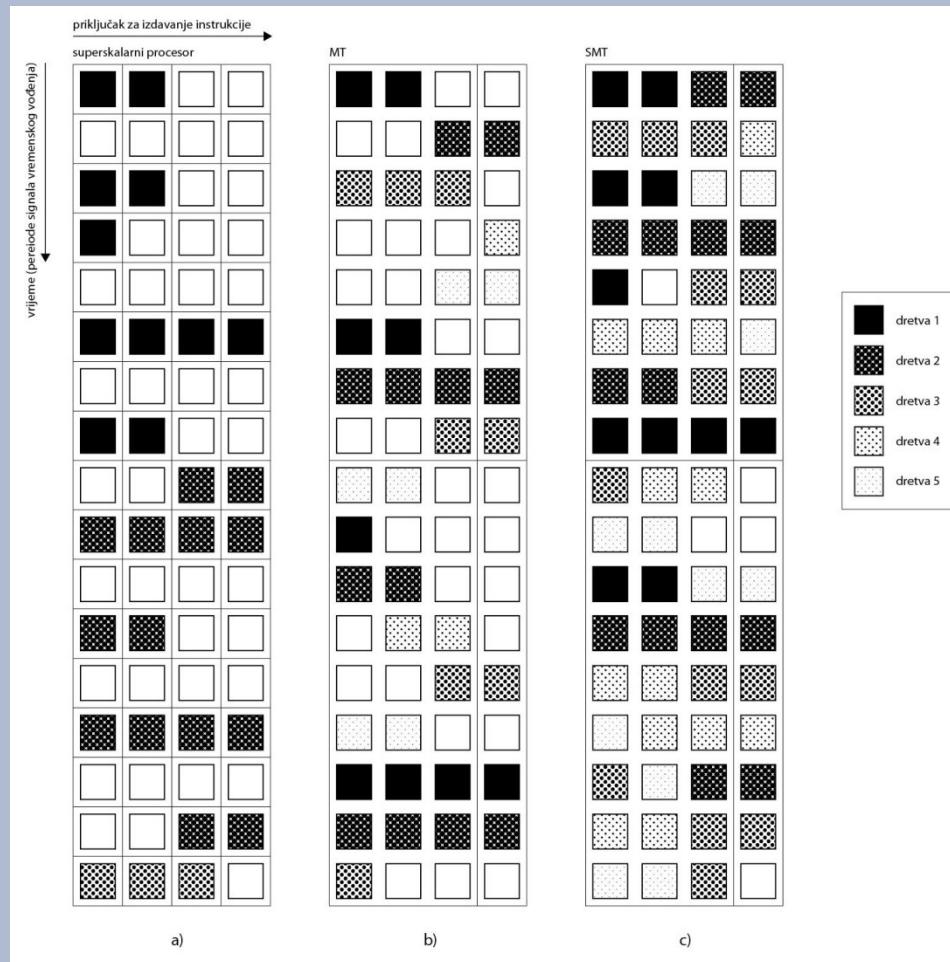
U SMT-u se iskorištava **paralelizam na razini dretvi i paralelizam na instrukcijskoj razini u kombinaciji s izdavanjem više instrukcija u jednoj periodi signala vremenskog vođenja.**

Primjeri procesora koji se temelje na SMT zamislima su Intelovi višejezgreni procesori Nehalem i Pentium D te IBM-ovi višejezgreni procesori Power 5, Power 6 i Power 7.

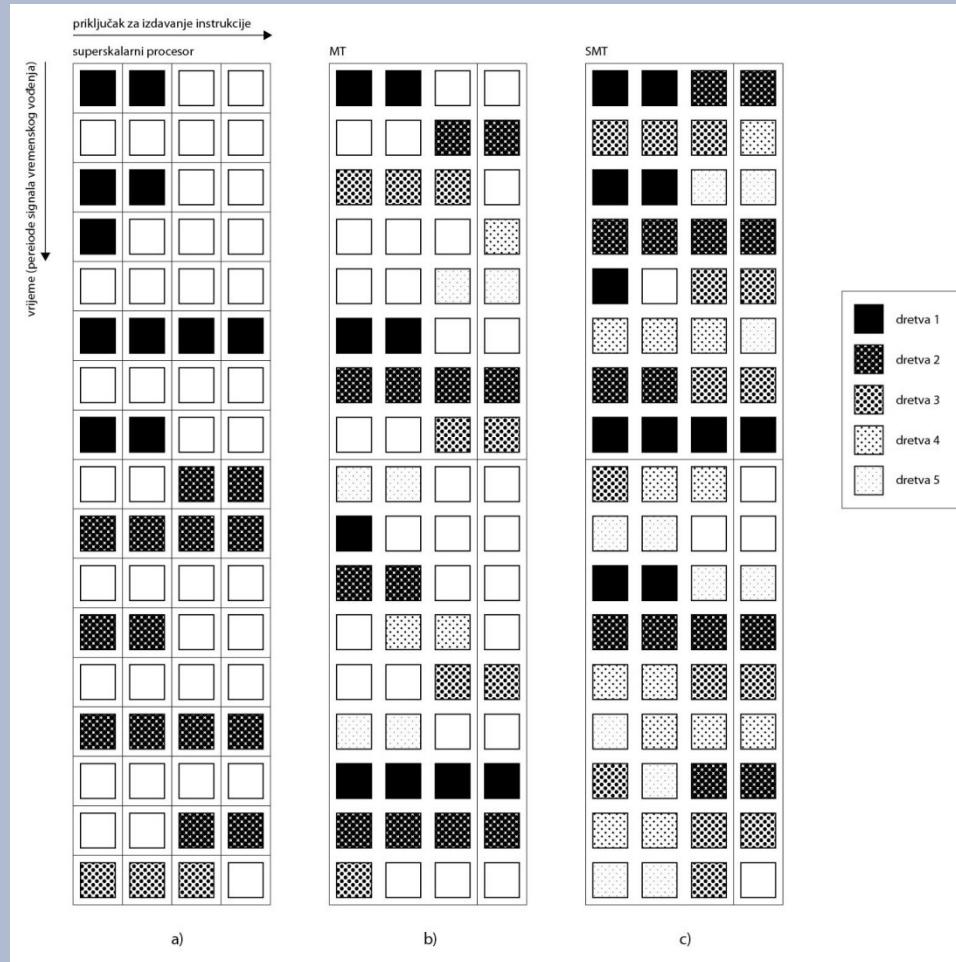
- razlika između superskalarnosti, višedretvenosti (MT) i simultane višedretvenosti (SMT), poslužit ćemo se jednostavnim modelom obrade koji su predložili S. J. Eggers i suradnici.



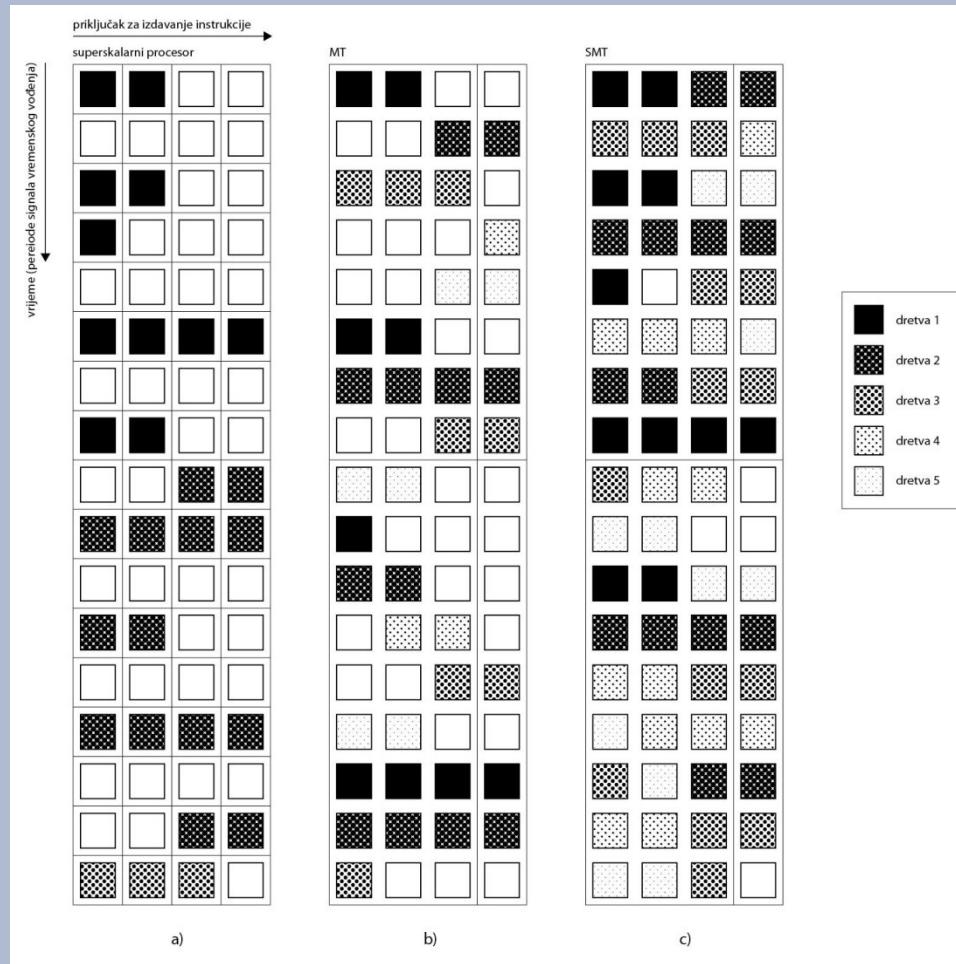
x os – priključci za izdavanje instrukcija  
y os – vrijeme (periode signala vremenskog vođenja)



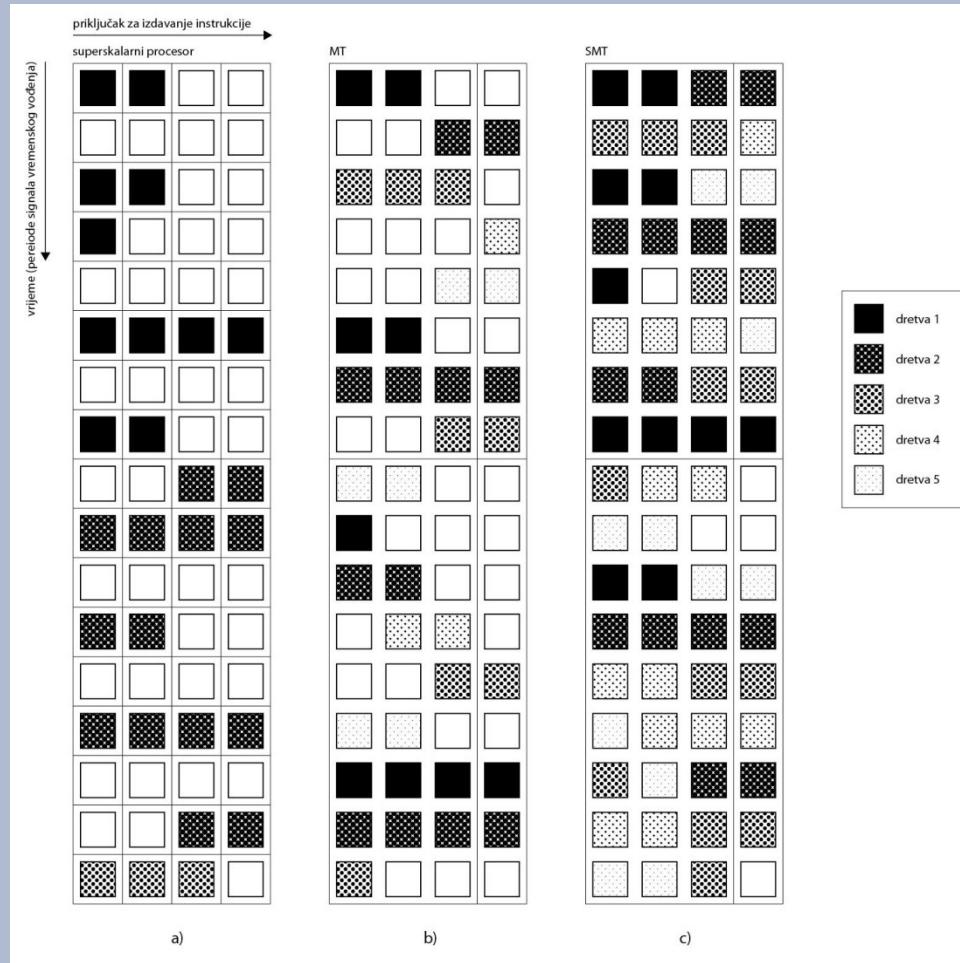
Nekorišteni priključci za izdavanje instrukcija mogu se promatrati kao "horizontalno neiskorišteni priključci" i kao "vertikalno neiskorišteni priključci"



Pod **horizontalno neiskorištenim priključcima** podrazumijevamo slučaj kada su u jednom retku jedan ili više priključaka neiskorišteni (ali ne svi!).



Vertikalno neiskorišteni priključci događaju se kada su tijekom jedne periode **svi priključci neiskorišteni** – to nastupa zbog dulje latencije (duljeg zastoja u izvođenju) instrukcije, npr. pristupa memoriji, kada je privremeno spriječeno daljnje izdavanje instrukcija.



Za SMT vidimo da u svakoj periodi signala vremenskog vođenja procesor "izabire" instrukcije za **izvođenje iz svih dretvi**.

- iskorištava se paralelizam na razini instrukcija izborom instrukcija iz bilo koje dretve.
- Nakon toga procesor dinamički raspoređuje resurse procesora između instrukcija i osigurava visoku razinu iskoristivosti sklopoških resursa.
- Ako neka od dretvi ima visok stupanj paralelizma na razini instrukcija, onda se te instrukcije izvode i pritom je vrlo malo horizontalno neiskorištenih priključaka.

- Ako paralelizam na razini instrukcija za jednu dretvu nije dovoljno visok, onda se izabire još jedna dretva (ili više njih) s nižim stupnjem paralelizma koja će popuniti prazne horizontalne priključke. Na taj se način postiže uklanjanje i vertikalnih neiskorištenih priključaka i u velikoj mjeri horizontalnih neiskorištenih

Pretpostavite da je procesor **superskalarni konvencionalne arhitekture** i da može izdavati do četiri instrukcije u jednoj periodi signala vremenskog vođenja.

Pretpostavite da protočne strukture nisu specijalizirane i da se trebaju izvesti sljedeće tri dretve prikazane na slici:

dretva1:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
|   |   |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |

dretva2:

|   |   |   |  |
|---|---|---|--|
| 2 | 2 | 2 |  |
| 2 | 2 |   |  |
| 2 |   |   |  |
| 2 |   |   |  |
| 3 | 3 | 3 |  |
| 3 |   |   |  |
| 3 |   |   |  |
| 3 | 3 | 3 |  |

dretva3

|   |   |   |  |
|---|---|---|--|
| 3 | 3 | 3 |  |
|   |   |   |  |
|   |   |   |  |
|   |   |   |  |
| 3 | 3 | 3 |  |

rješenje:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
|   |   |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 2 | 2 | 2 |   |
| 2 | 2 |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
| 2 | 2 | 2 | 2 |
| 2 | 2 |   |   |
| 2 | 2 |   |   |
| 3 | 3 | 3 |   |
| 3 |   |   |   |
| 3 |   |   |   |
| 3 | 3 | 3 |   |

Za tri dretve prikažite izvođenje za **višedretveni superskalarni procesor** (izdaje do četiri instrukcije) koji koristi **finozrnatu dretvenost**.

Prepostavite da protočne strukture nisu specijalizirane i da se trebaju izvesti sljedeće tri dretve prikazane na slici:

dretva1:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |

dretva2:

|   |   |   |   |
|---|---|---|---|
| 2 | 2 | 2 |   |
| 2 | 2 |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
| 2 | 2 | 2 | 2 |
| 2 | 2 |   |   |
| 2 | 2 |   |   |

dretva3:

|   |   |   |  |
|---|---|---|--|
| 3 | 3 | 3 |  |
| 3 | 3 |   |  |
| 3 |   |   |  |
| 3 |   |   |  |
| 3 |   |   |  |
| 3 | 3 | 3 |  |
| 3 |   |   |  |
| 3 |   |   |  |

rješenje:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 2 | 2 | 2 |   |
| 3 | 3 | 3 |   |
| 1 |   |   |   |
| 2 | 2 |   |   |
| 3 | 3 |   |   |
| 1 | 1 | 1 |   |
| 2 |   |   |   |
| 3 |   |   |   |
| 1 | 1 |   |   |
| 2 |   |   |   |
| 3 |   |   |   |
| 1 | 1 | 1 | 1 |
| 2 |   |   |   |
| 3 | 3 | 3 |   |
| 1 |   |   |   |
| 2 | 2 | 2 | 2 |
| 1 | 1 | 1 |   |
| 2 | 2 |   |   |
| 1 | 1 |   |   |
| 2 | 2 |   |   |

Finozrnata višedretvenost podrazumijeva prospajanje dretvi nakon svake instrukcije. Te su instrukcije međusobno nezavisne jer pripadaju različitim dretvama tako se protočna struktura djelotvorno iskorištava.

Prikažite izvođenje za **višedretveni superskalarni procesor** (izdaje do četiri instrukcije) koji koristi **grubozrnatu dretvenost**. Pretpostavite da se prospajanje dretvi događa kod zastoja koji odgovara vertikalno neiskorištenim priključcima. Također pretpostavite da prospajanje dretvi u ovom slučaju zahtijeva jednu periodu signala vremenskog vođenja.

rješenje:

dretva1:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
|   |   |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |

dretva2:

|   |   |   |   |
|---|---|---|---|
| 2 | 2 | 2 |   |
| 2 | 2 |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
| 2 | 2 | 2 | 2 |
| 2 | 2 |   |   |
| 2 | 2 |   |   |

dretva3:

|   |   |   |  |
|---|---|---|--|
| 3 | 3 | 3 |  |
| 3 | 3 |   |  |
| 3 |   |   |  |
| 3 | 3 | 3 |  |
| 3 | 3 |   |  |

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
|   |   |   |   |
| 2 | 2 | 2 |   |
| 2 | 2 |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
| 2 | 2 | 2 | 2 |
| 2 | 2 |   |   |
| 2 | 2 |   |   |
| 3 | 3 | 3 |   |
| 3 | 3 |   |   |
| 3 |   |   |   |
| 3 | 3 | 3 |   |
| 3 | 3 |   |   |

Prikažite izvođenje u superskalarnom simultanom višdretvenom procesoru SMT (izdaje do četiri instrukcije) uz pretpostavku da protočne strukture nisu specijalizirane.

rješenje:

dretva1:

|   |   |   |   |
|---|---|---|---|
| 1 | 1 |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |
| 1 | 1 | 1 | 1 |
|   |   |   |   |
|   |   |   |   |
| 1 |   |   |   |
| 1 | 1 | 1 |   |
| 1 | 1 |   |   |

dretva2:

|   |   |   |   |
|---|---|---|---|
| 2 | 2 | 2 |   |
| 2 | 2 |   |   |
| 2 |   |   |   |
| 2 |   |   |   |
|   |   |   |   |
|   |   |   |   |
| 2 | 2 | 2 | 2 |
| 2 | 2 |   |   |
| 2 | 2 |   |   |

dretva3:

|   |   |   |  |
|---|---|---|--|
| 3 | 3 | 3 |  |
|   |   |   |  |
| 3 | 3 |   |  |
| 3 |   |   |  |
| 3 | 3 | 3 |  |
|   |   |   |  |
|   |   |   |  |

|   |   |   |   |
|---|---|---|---|
| 1 | 1 | 2 | 2 |
| 2 | 3 | 3 | 3 |
| 1 | 2 | 2 |   |
| 1 | 1 | 1 | 2 |
| 3 | 3 | 1 | 1 |
| 2 | 3 | 1 | 1 |
| 2 | 3 | 1 | 1 |
| 3 | 3 | 3 |   |
| 2 | 2 | 2 | 2 |
| 1 | 2 | 2 |   |
| 1 | 1 | 1 | 2 |
| 1 | 1 | 2 |   |