

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 μ prog)
- **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} :

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

Pristup oblikovanja puta podataka arhitekture RISC:

- $\text{CPI} \approx 1$, 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, instrukcije imaju sličnu složenost)
- Prioritet: maksimalno ubrzati najčešću operaciju:
add r1, r2, r3

MIPS: primjer procesora arhitekture RISC

1. Skup registara

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

Broj registra	Mnemoničko ime	Funkcija
\$0	\$zero	Konstantna vrijednost 0
\$1	\$at	Rezerviran za assembler
\$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) Memorijske
 - (B) Aritmetičko – logičke
 - (C) Instrukcije grananja
- Dodatno:
 - instrukcije za rad s brojevima s pomičnim zarezom (FP) – ne razmatramo

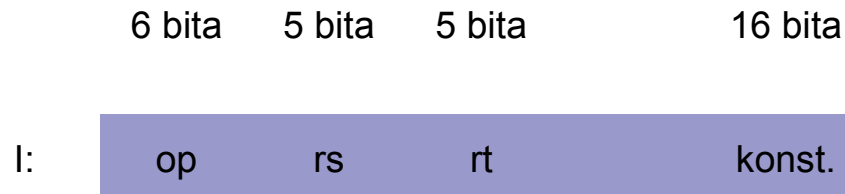
2. MIPS – skup instrukcija:

(A) Memorijske 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**":



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

2. MIPS – skup instrukcija:

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

npr.

```
add $s1, $s2, $s3      // add; $s1 ← $s2 + $s3
sub $s1, $s2, $s3      // subtract; $s1 ← $s2 - $s3
and $s1, $s2, $s3      // and; $s1 ← $s2 & $s3
or $s1, $s2, $s3       // or; $s1 ← $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.) **predznačno proširene na 32 bita**
- Rezultat se sprema u **odredišni registar rt**

2. MIPS – skup instrukcija:

(C.1) Instrukcije grananja s apsolutnom adresom

Bezuvjetno apsolutno 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 registar **\$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 $PC \leftarrow (PC+4) + 25 \ll 2$)

```
bne $t1, $t2, 25 // branch if not equal;
```

(ako je $\$t1 \neq \$t2$ onda $PC \leftarrow (PC+4) + 25 \ll 2$)

- **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**

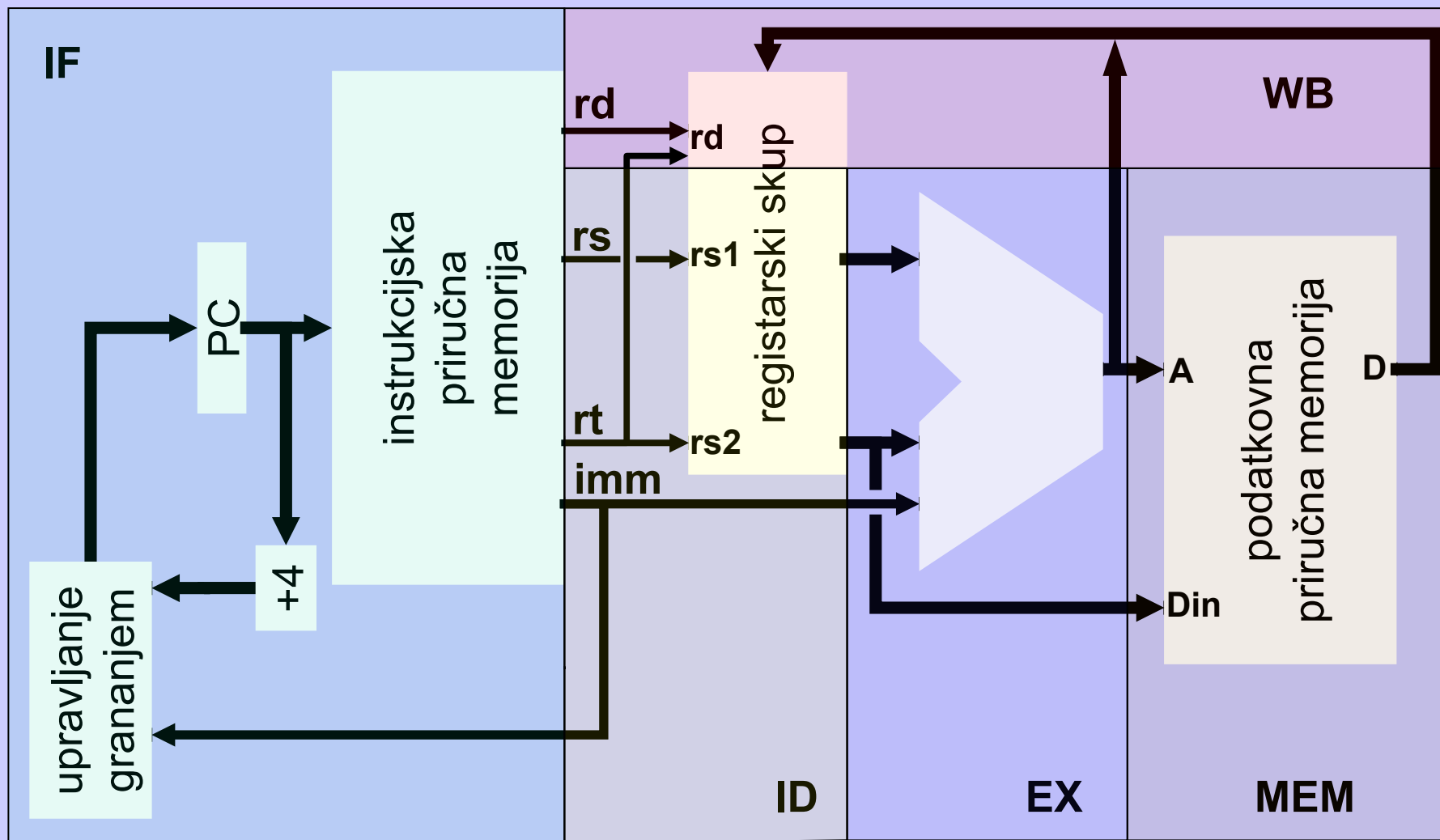
Tijek izvođenja instrukcija iz tri najčešće grupe:

- pribavljanje 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!)
- memorijske 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 memorijske instrukcije)
5. WB: upis odredišnog registarskog operanda
(aritmetika, load)

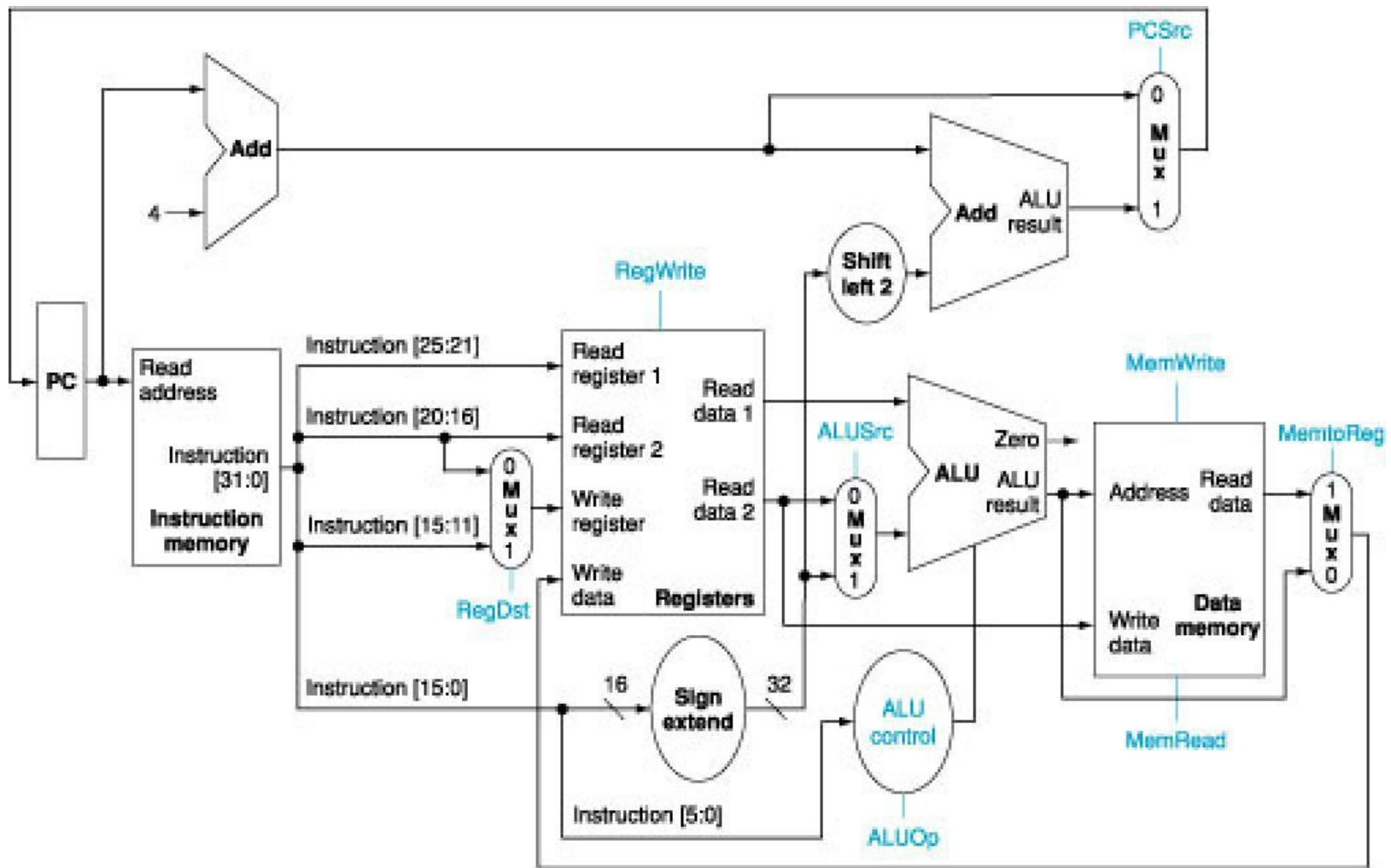
Put podataka za računalno arhitekturu MIPS - pojednostavnjeno



Detaljan prikaz puta podataka arhitekture MIPS

- za minimalni podskup skupa instrukcija:

lw, sw, beq, add, sub, and, or, slt (I- i R- tip)

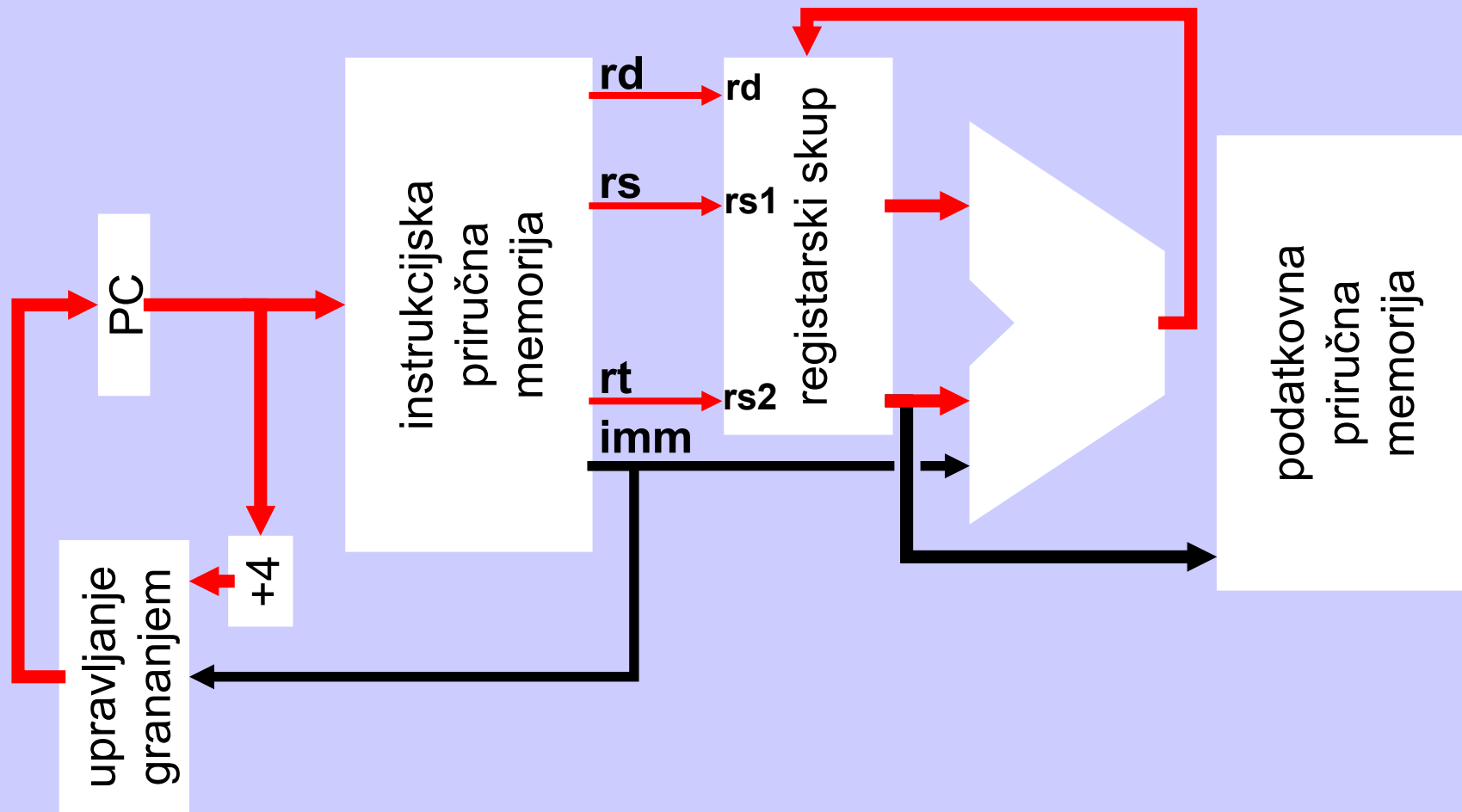


Primjer 1: registarska aritmetička instrukcija

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

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

sub \$rd, \$rs,\$rt # \$rd←\$rs-\$rt



R-tip

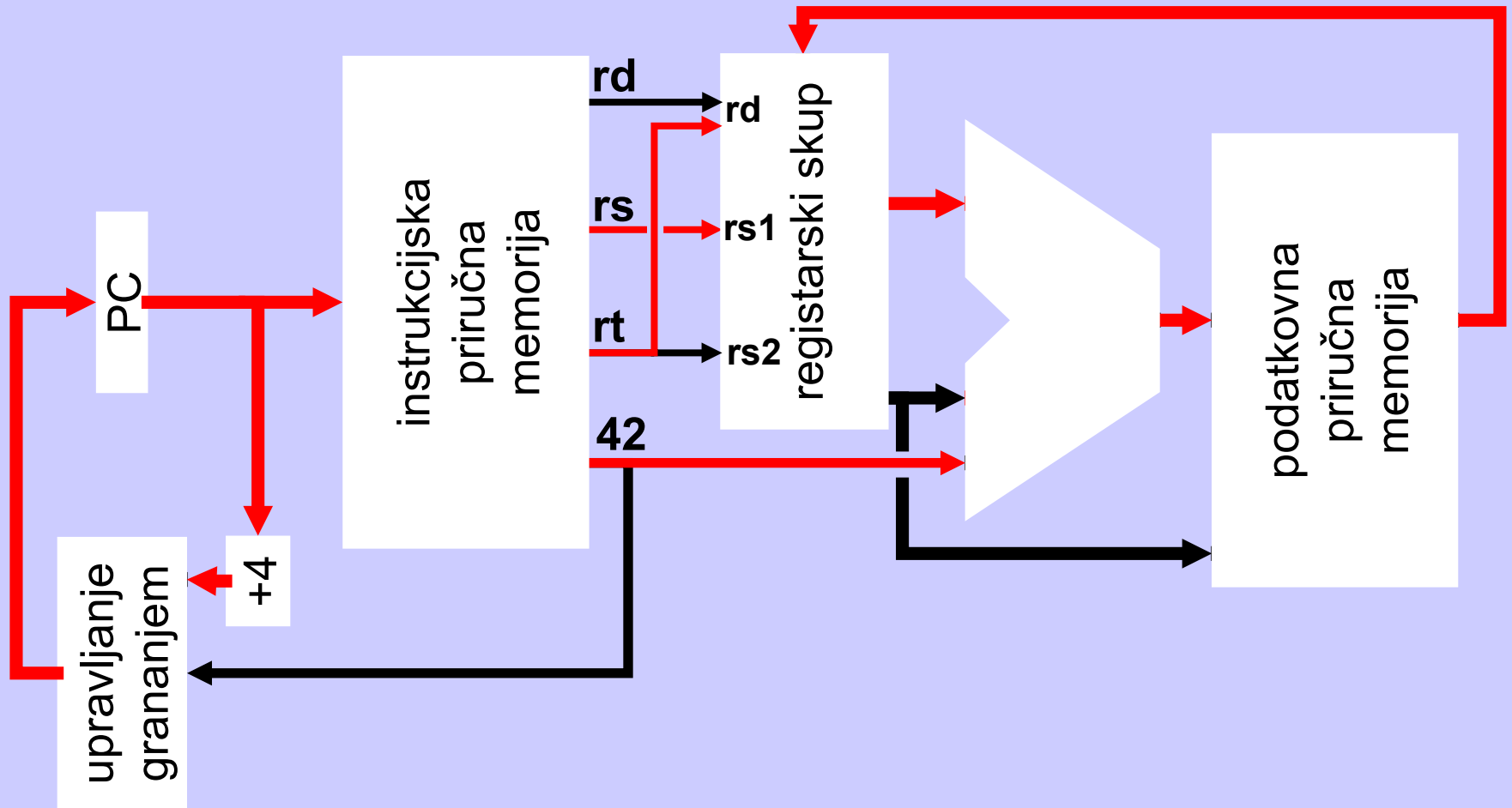
000000 rs rt rd 000000 100011

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) # $rt ← MEM[$rs+42]`



I-tip

100011

rs

rt

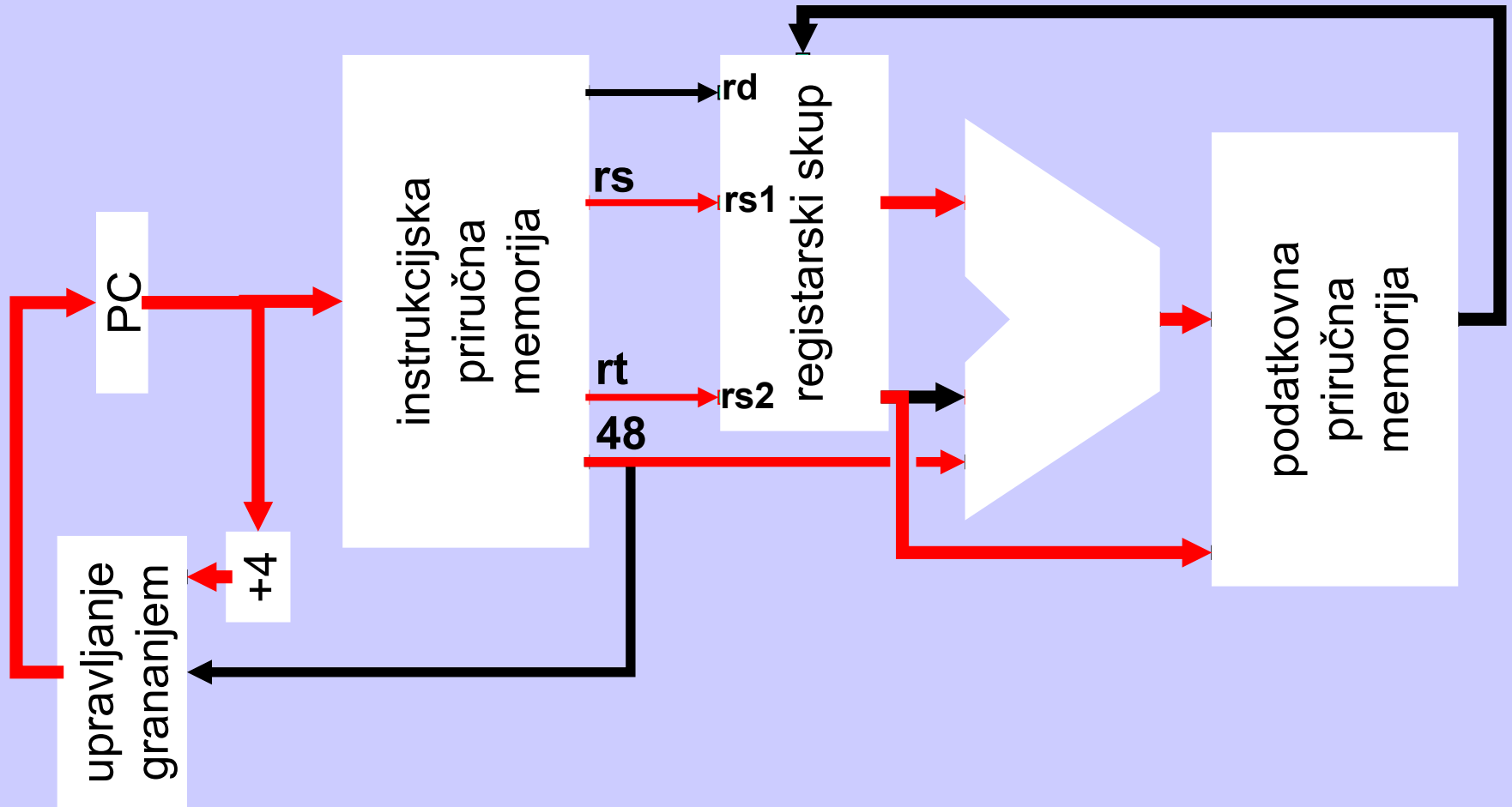
0000 0000 0100 0010

Primjer 3: memorijska instrukcija

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

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

sw \$rt, 48(\$rs) # MEM[\$rs+48]←\$rt



I-tip

101011

rs

rt

0000 0000 0100 1000

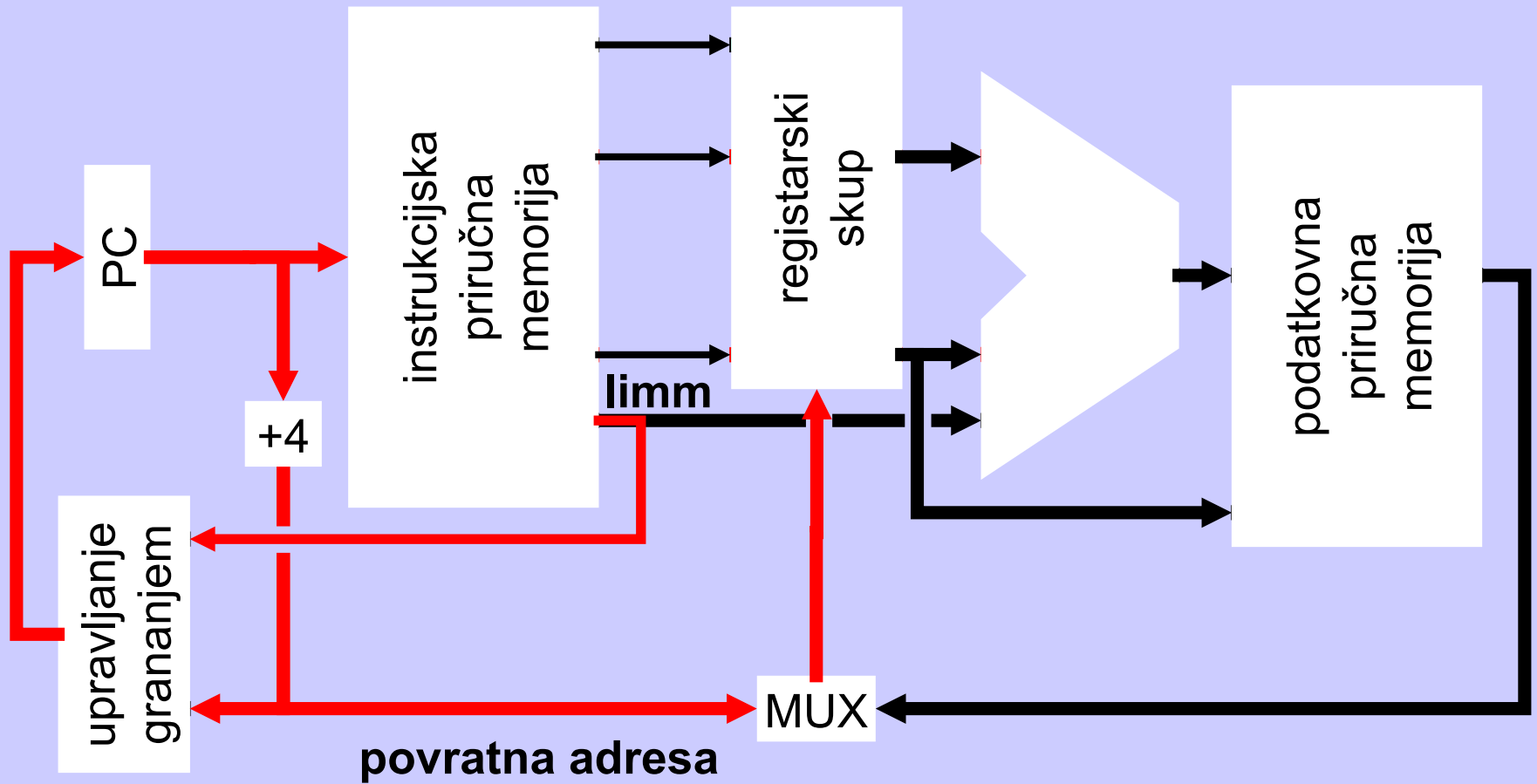
Primjer 4: instrukcija grananja:

```
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

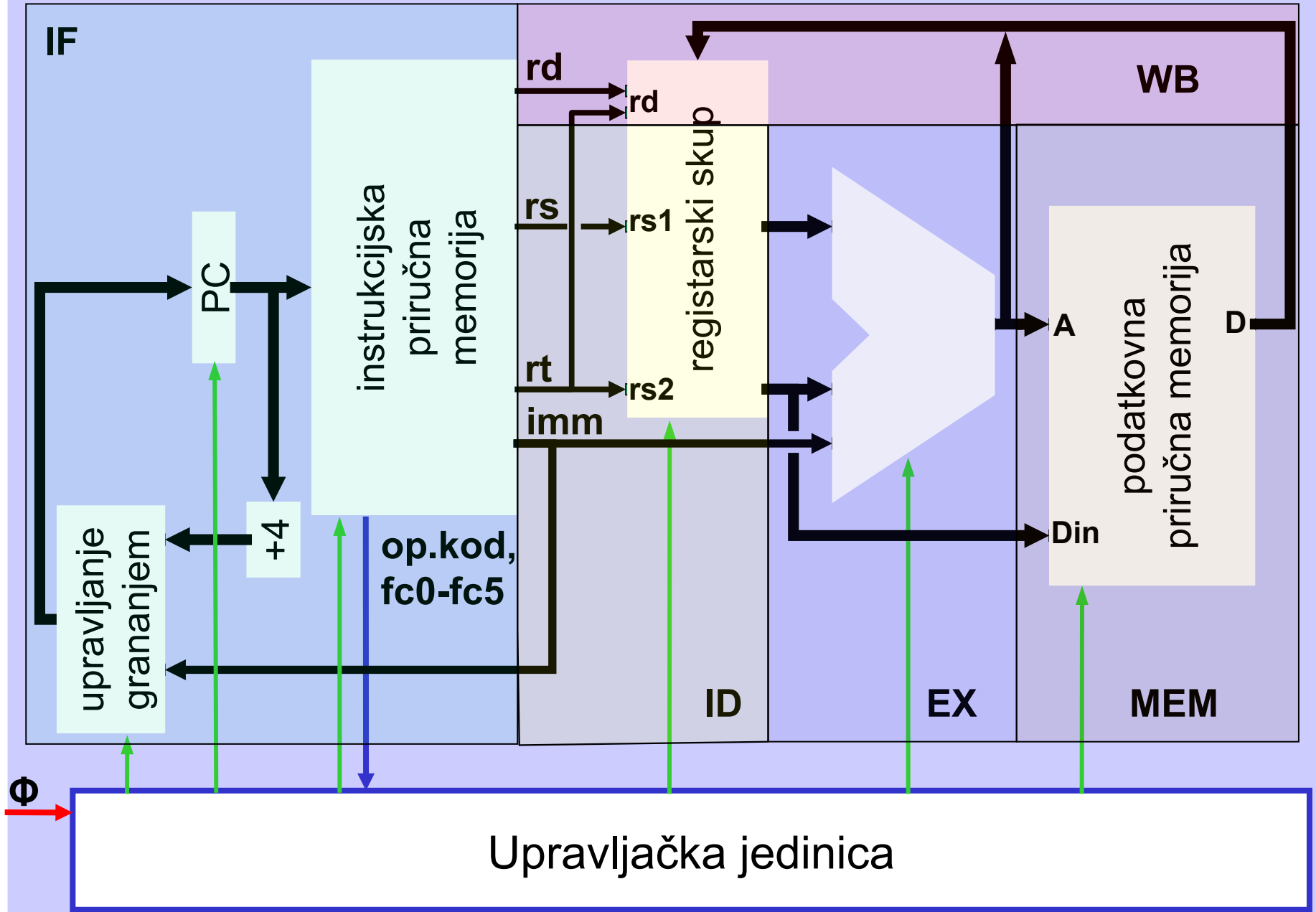
* u zakašnjeloj izvedbi PC pokazuje na priključak za kašnjenje

** limm se množi s 4!

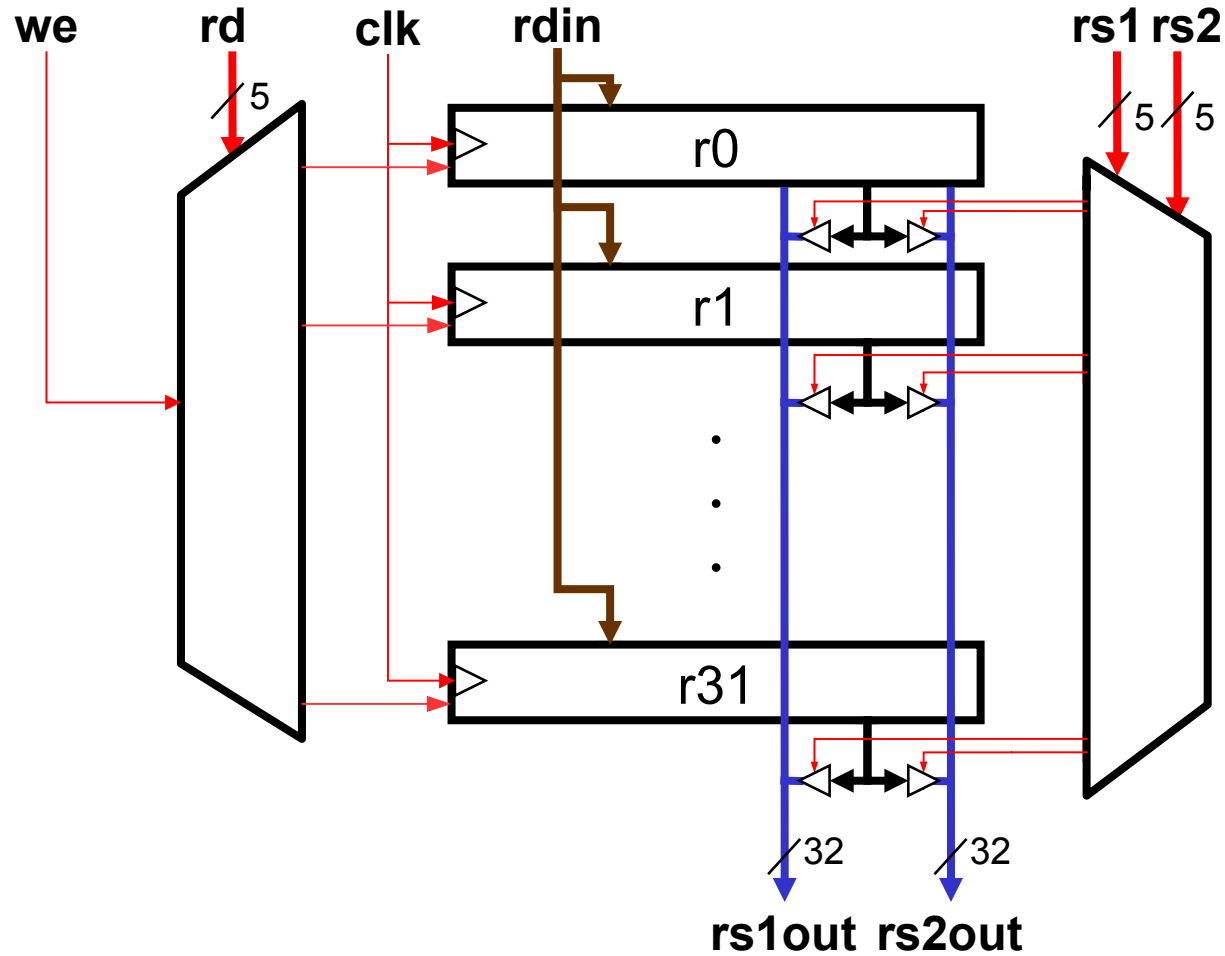
Svojstva puta podataka arhitekture MIPS:

- 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 sklopovlje:
 - dekoderi i multiplekseri
 - interne sabirnice
 - registarski 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

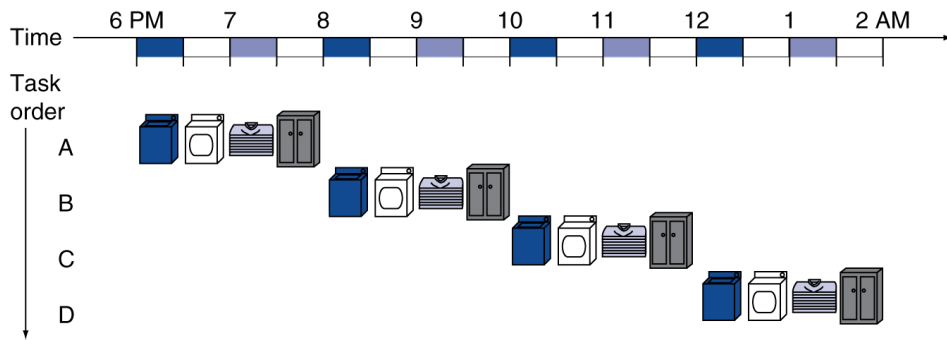


Pogled na performansu

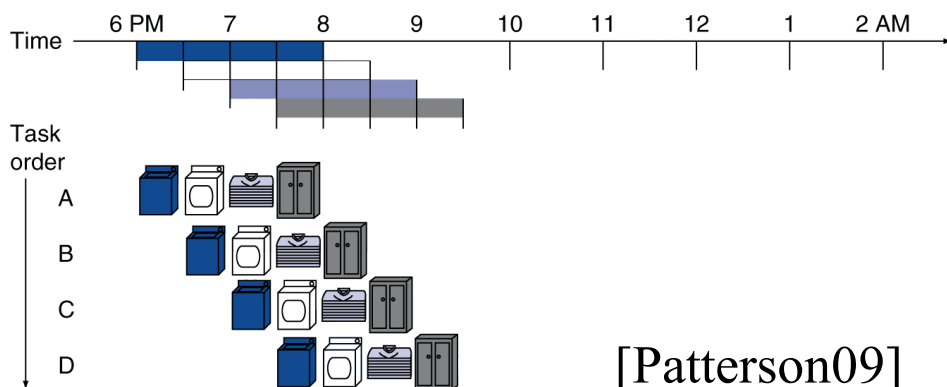
- trajanje instrukcije određuje najdulje kašnjenje
 - kritični put: instrukcija load
 - instrukcijska memorija → registri → ALU → podatkovna memorija → registri
- možemo li popraviti performansu bez skraćanja trajanja instrukcije?
 - da, ako uspijemo izvoditi **više instrukcija istovremeno**
 - potencijal **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:
ubrzanje = $8/3.5 = 2.3$



- kontinuirani rad (**n**):
ubrzanje = $2n/(0.5n+1.5) \approx 4$
= broj segmenata!

- ideja posuđena iz manufaktura i proizvodnih linija [Chaplin 1936]

[Patterson09]

Protočna organizacija arhitekture MIPS

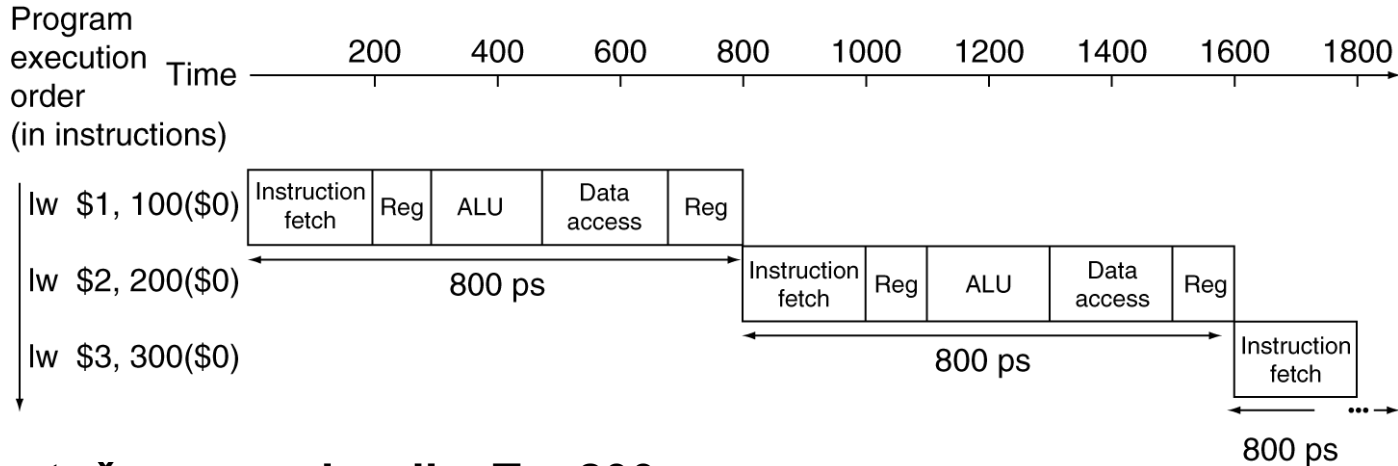
- pretpostavimo da pristup registrima traje 100ps, a ostale operacije 200ps
- tada dobivamo sljedeće latencije tipičnih instrukcija:

instrukcija	instrukcijska memorija	čitanje registara	zbrajanje	memorija podataka	upis registra	latencija (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

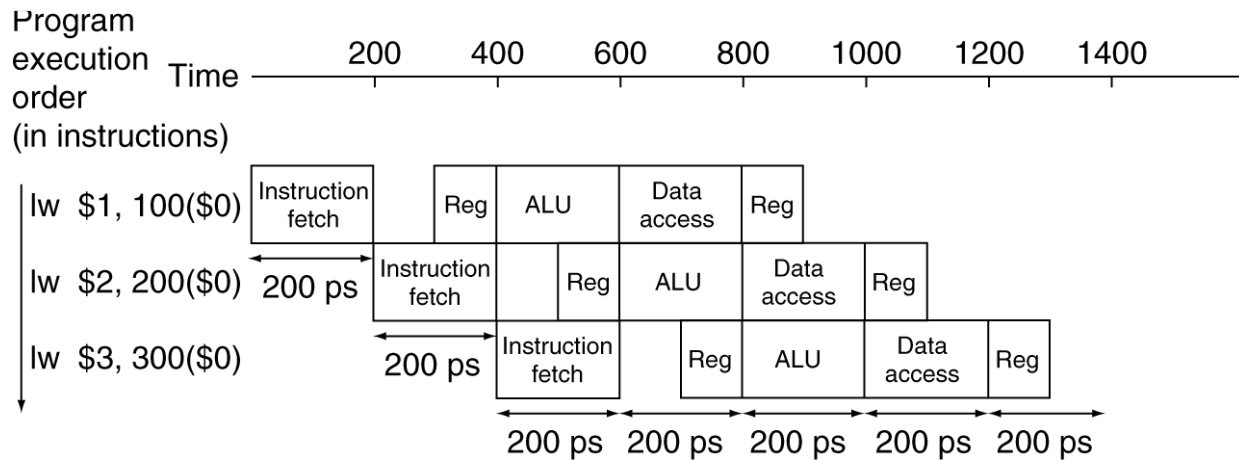
[Patterson09]

- 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 **x4!**

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, četvrti 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 postizemo 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
 - **međuovisnost 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

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

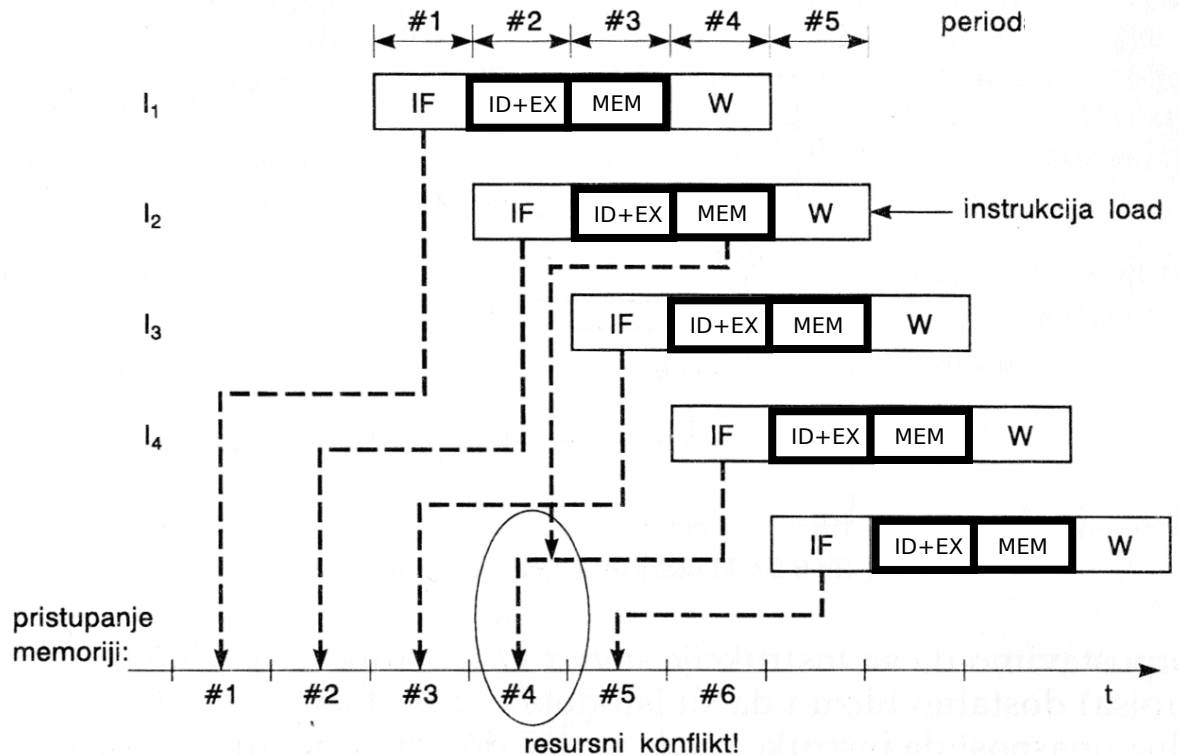
Tri razreda hazarda:

- strukturni hazard
- podatkovni hazard
- upravljački hazard

Strukturni hazard (resursni konflikt)

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

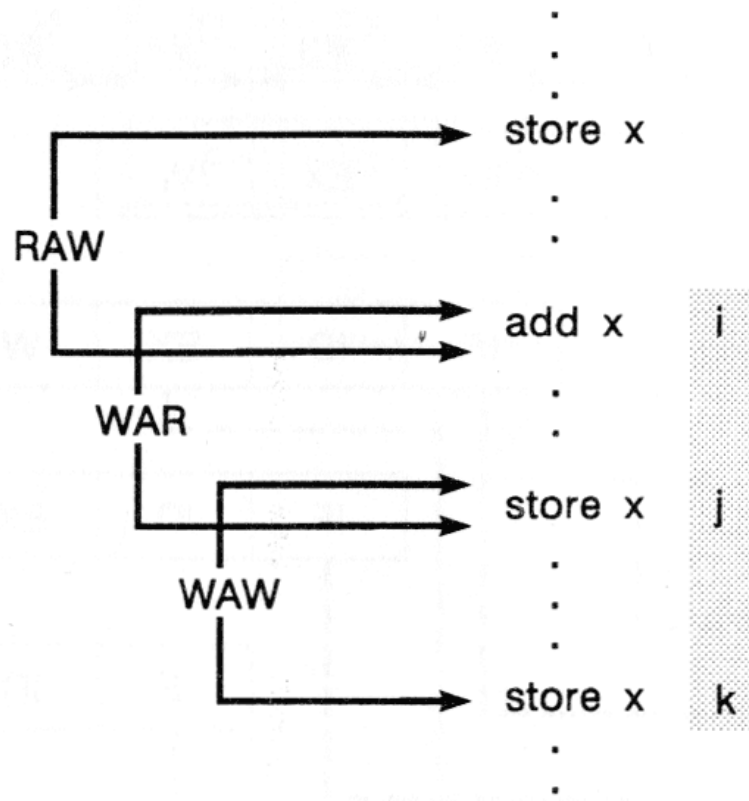
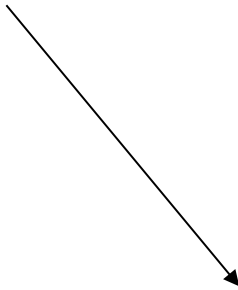
Primjer: konflikt kod pristupa zajedničkoj memoriji kod arhitekture Berkeley RISC (4 protočna segmenta):



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 određeni 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:

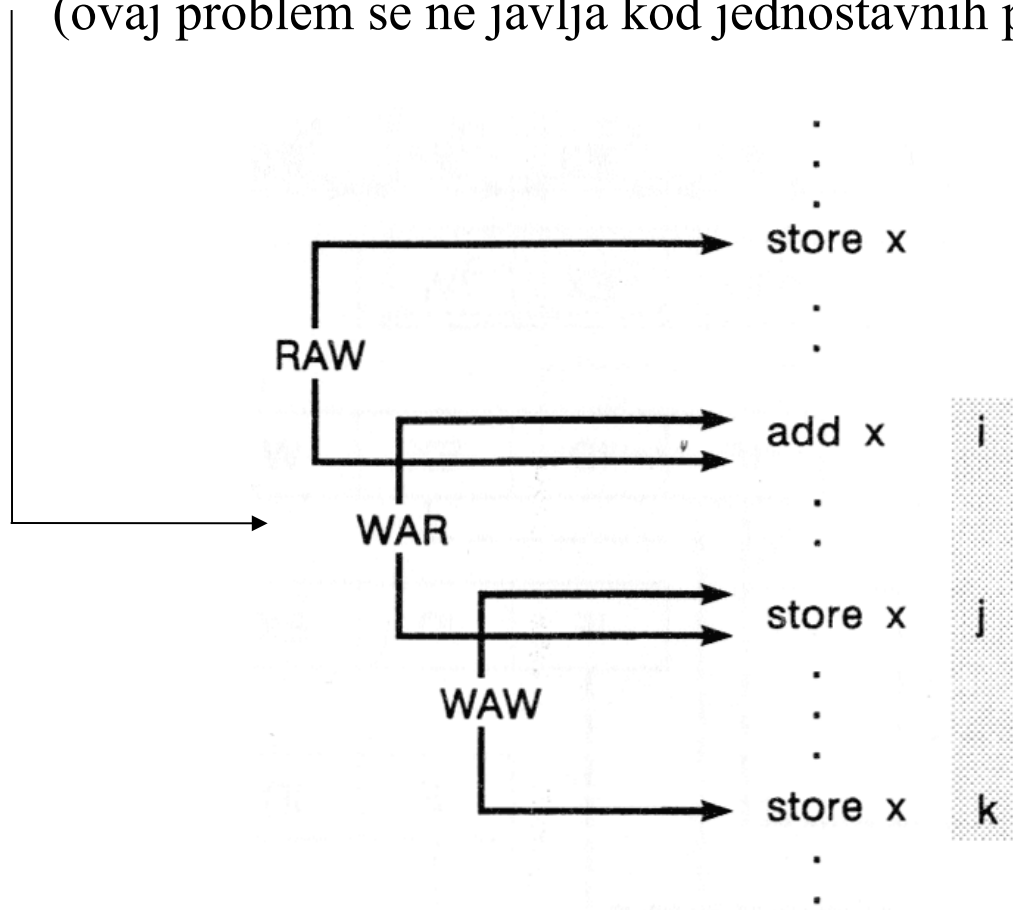
```
add $r1, $r2,$r3
```

```
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 protočnih arhitektura)



registarski primjer:

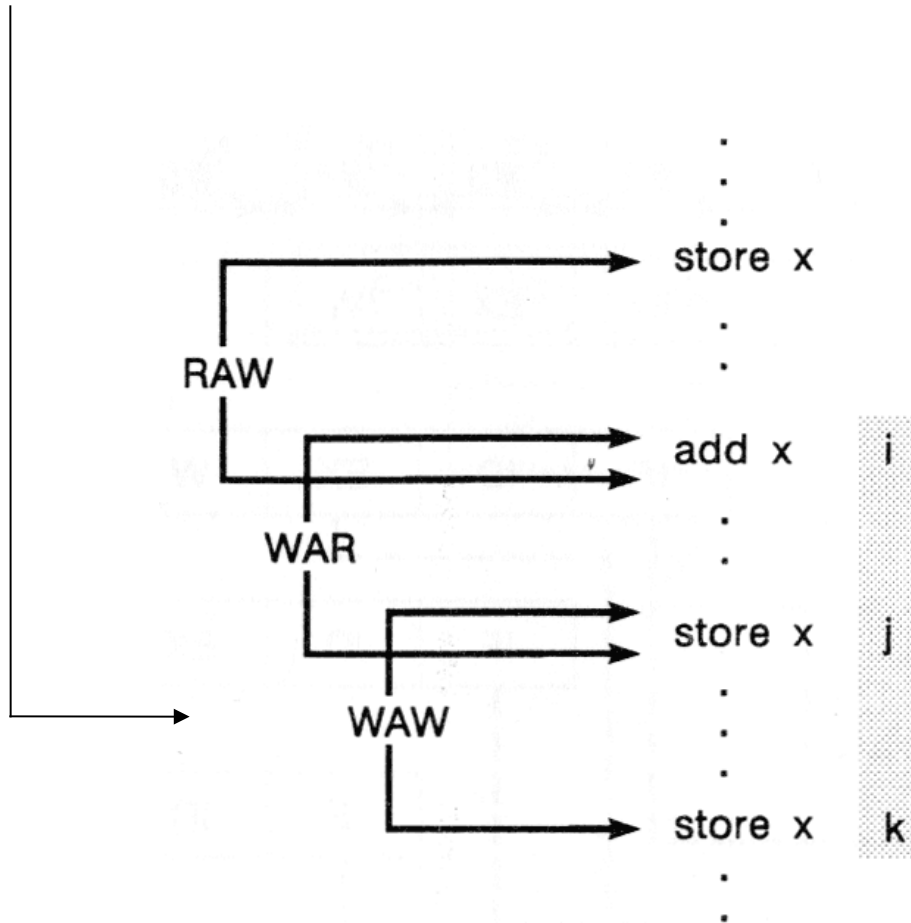
```
add $r1, $r2,$r3
```

```
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 protočnih arhitektura)



registarski primjer:

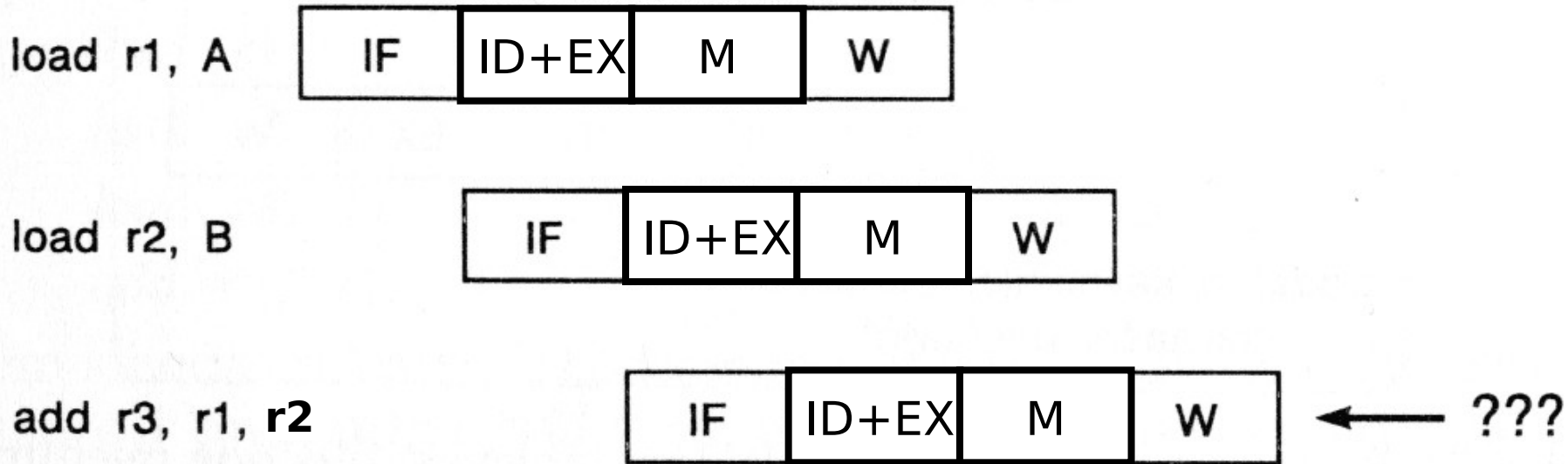
```
add $r1,$r2,$r3
```

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

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

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

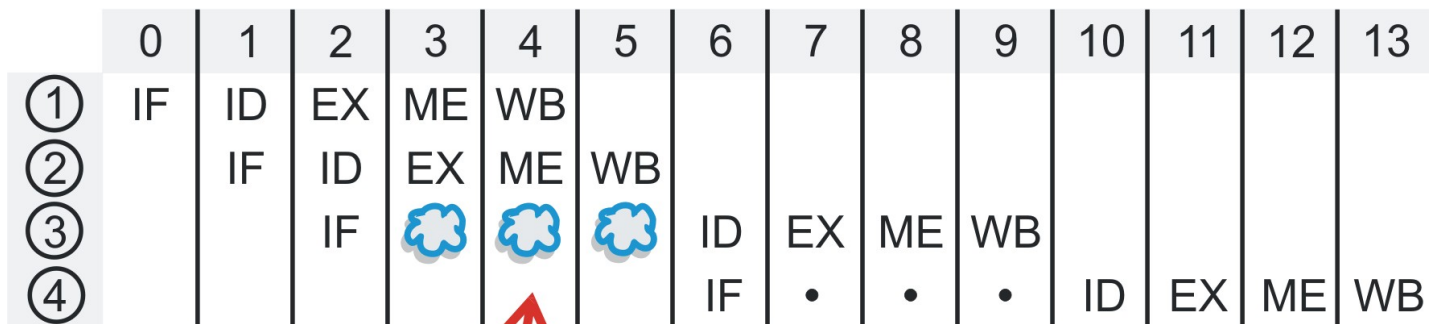

Hazard vrste RAW je u protočnim arhitekturama najviše izražen tijekom izvođenja instrukcije *load*



Saniranje hazarda RAW nakon instrukcije load:

- usporavanjem protočnosti (**protočni mjehurići**, vidi dolje)
- **prosljeđivanjem** npr, MEM[i] → EX[i+1] (ipak jedan mjehurić)
- specifičnom definicijom instrukcije load (**zakašnjelo čitanje**)

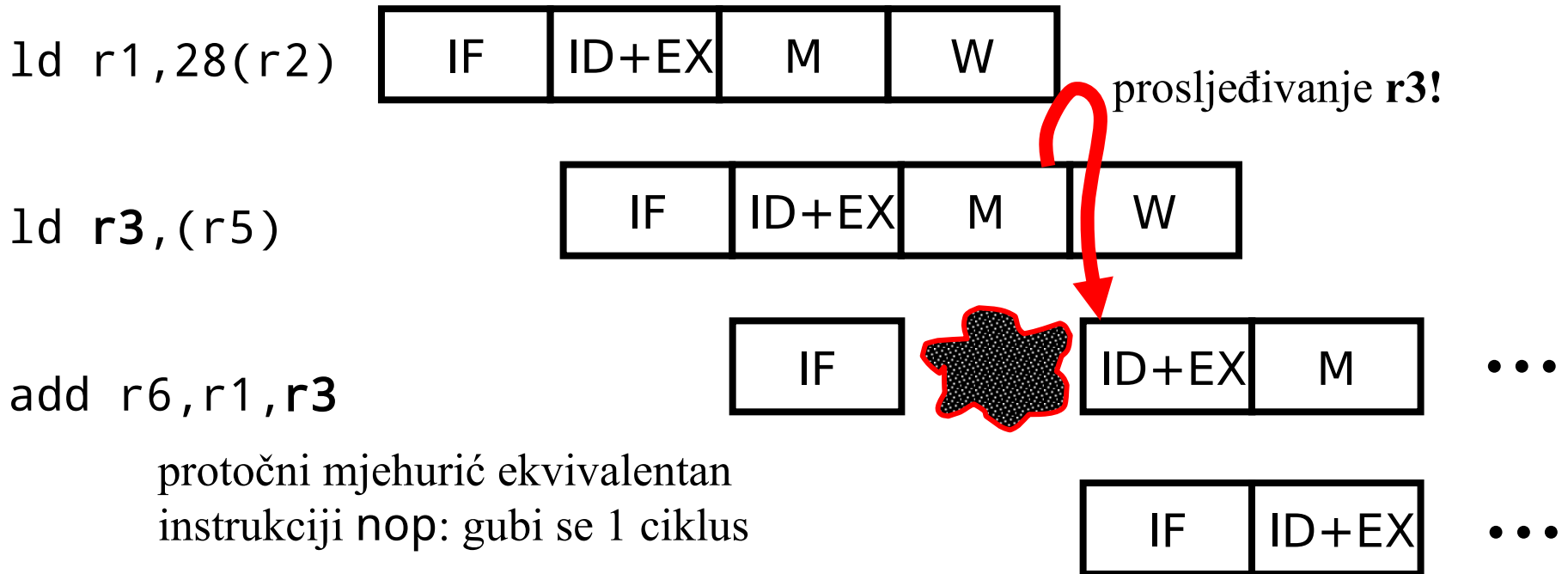
load $r5 = mem(r1 + r2)$
load $r6 = mem(r3 + offset)$
add $r7 = r5, r6$
store $mem(r1 + r2) = r7$.



protočni mjehurići

Kod jednostavnih instrukcija, najčešće možemo proći samo s jednim mjehurićem

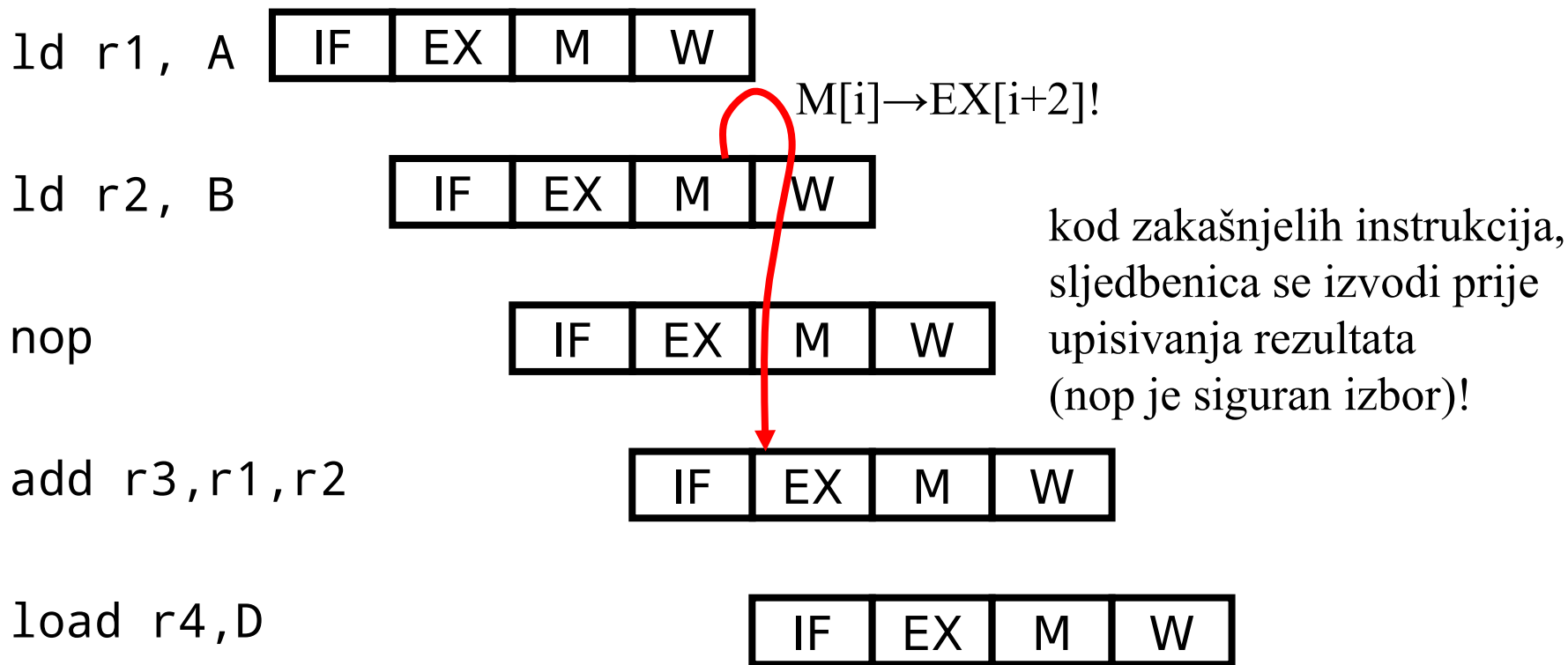
- **prosljeđivanje:** prospajanje (vodovi + MUX) rezultata prethodne operacije natrag prema ulazu procesne jedinice
- primjer za arhitekturu Berkeley RISC $M[i] \rightarrow ID+EX[i+1]$:



Ideja: potpuno otkloniti potrebu za mjehurićima **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.



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*

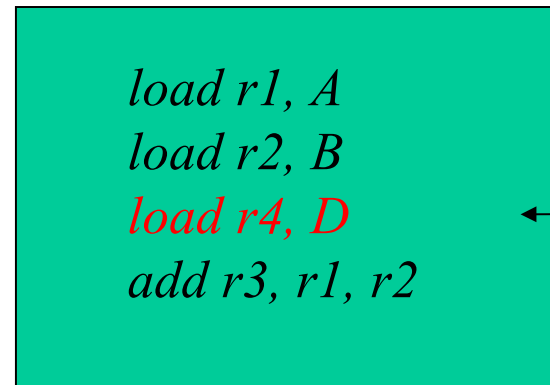
Instrukcija iz priključka za kašnjenje ne vidi rezultate izvođenja prethodne instrukcije: logički, dvije instrukcije se izvršavaju **istovremeno**

Primjer (zakašnjelo čitanje):

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

load r1, A
load r2, B
add r3, r1, r2
load r4, D

← hazard RAW

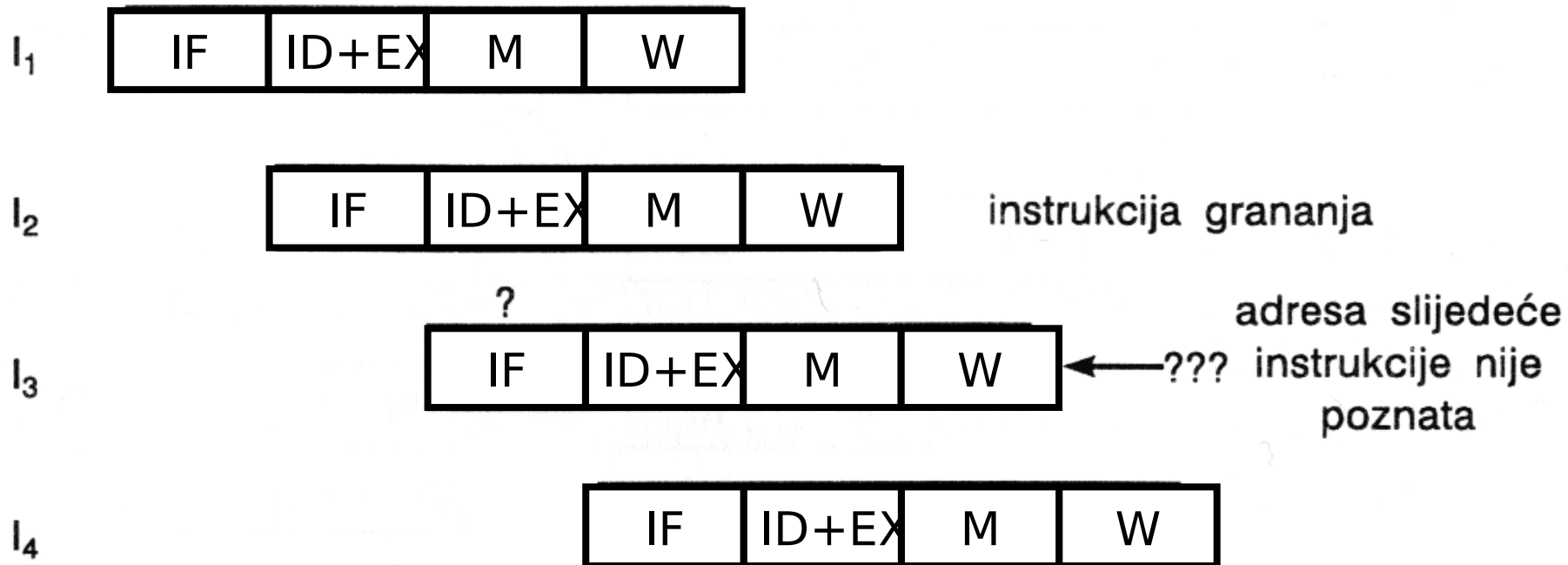


← priključak

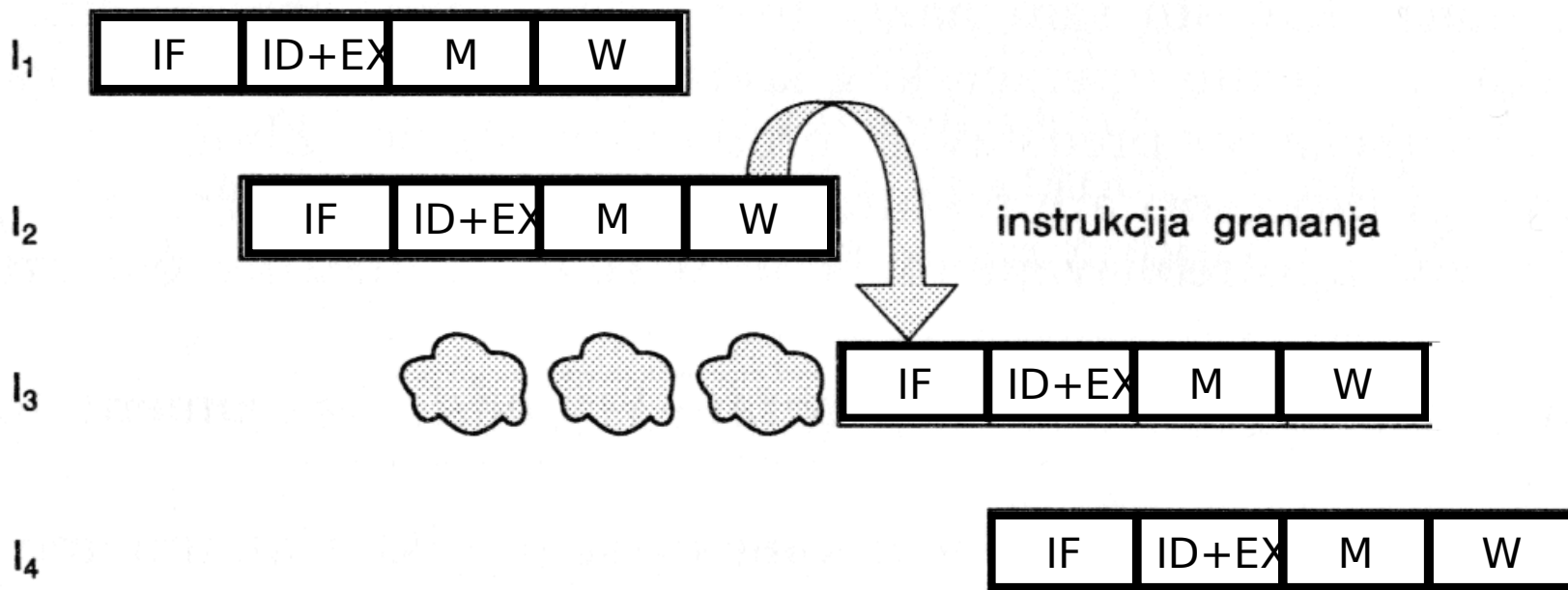
Upravljački hazard

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 (!!)



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 $PC+4+imm$

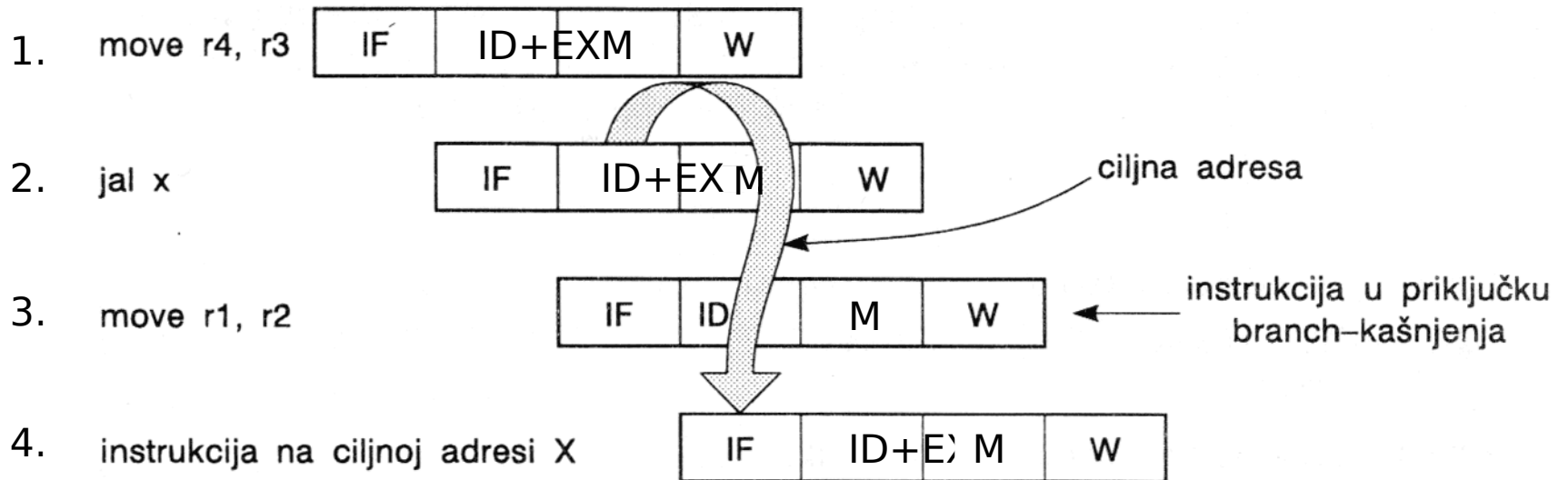
Kašnjenje samo s jednim mjehurićem!

Mjehurići se mogu potpuno zaobići **zakašnjelim grananjem**

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

Zakašnjelo grananje:

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



Primjer za računalo MIPS: uvjetno relativno grananje

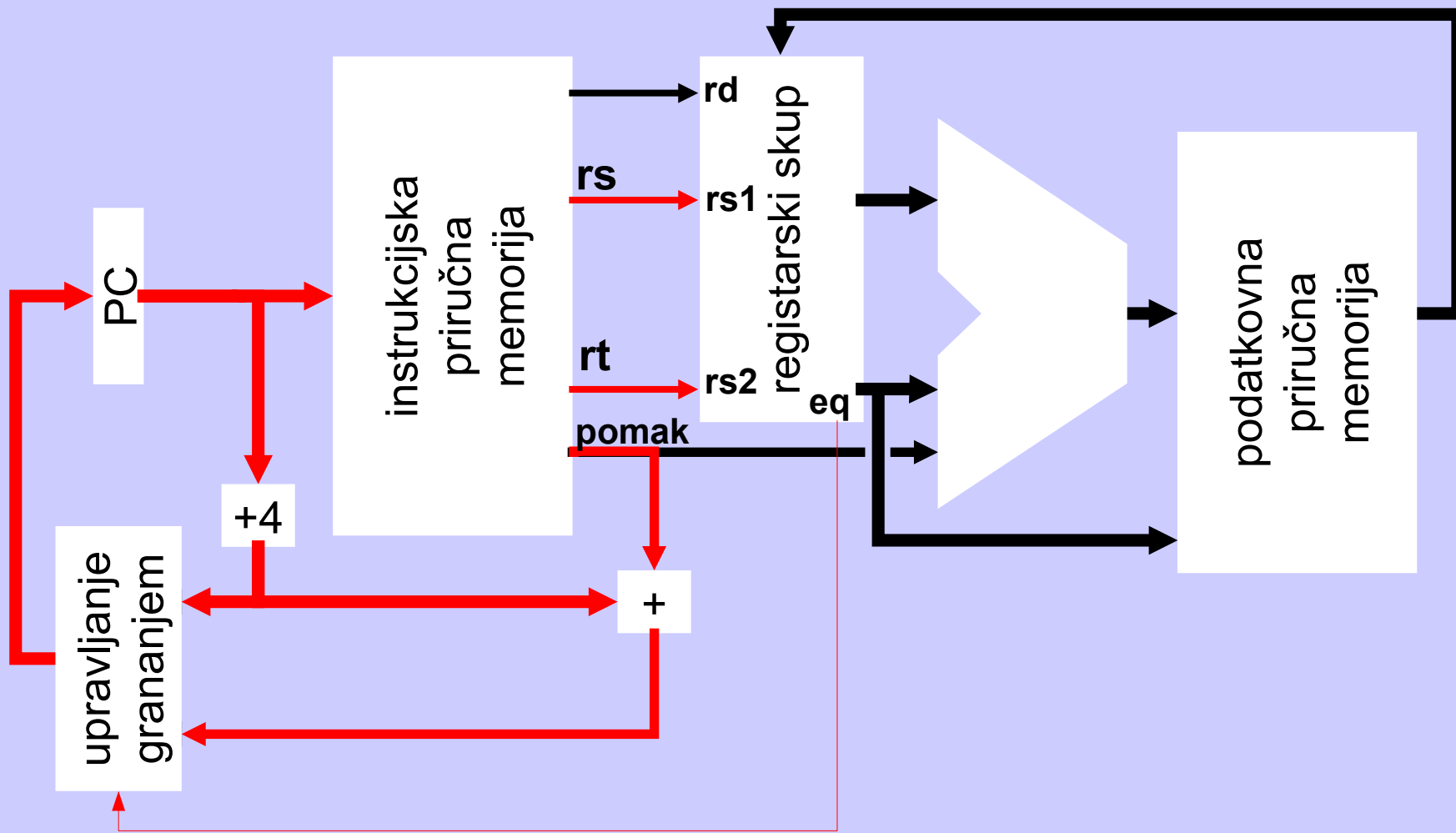
bne \$r1, \$r2, pomak # PC ← PC + 4 + pomak, ako ($r1 \neq r2$)

- 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+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←PC+4+pomak, ako (\$r1≠\$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 jednako 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
 - μ operacije po segmentima jednako 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.