7. Organizacija modernog procesora tipa SISD:

- 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≈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!)

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
- memorijske instrukcije pristupaju memoriji
- upisivanje rezultata u odredišni registar

Segmenti puta podataka arhitekture MIPS:

1. IF: pribavljanje 32-bitne instrukcije, PC=PC+4

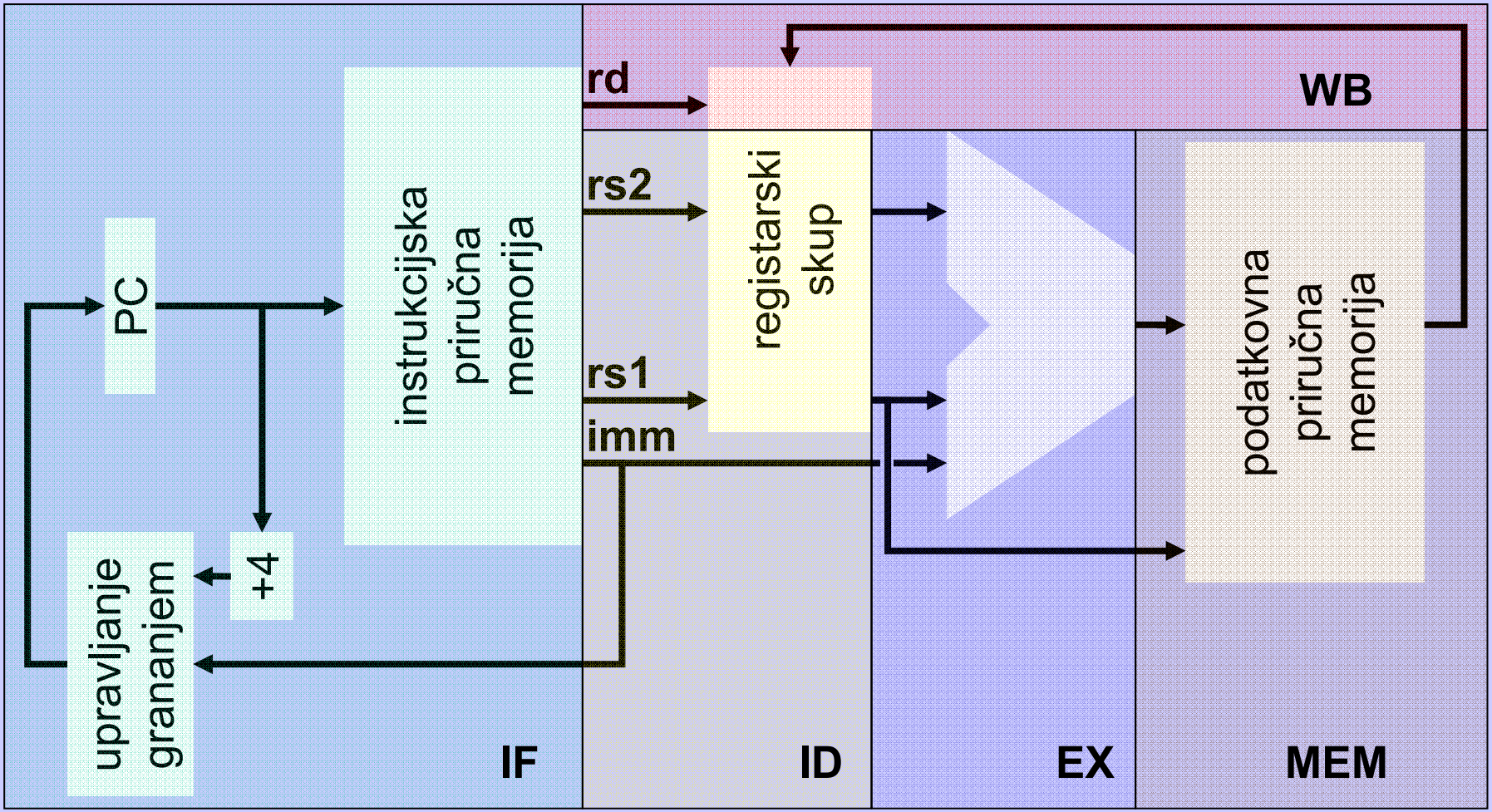
2. ID: dekodiranje operacijskog koda, pribavljanje registarskih operanada (0,1 ili 2)

3. EX: zbrajanje ili posmak
   (aritmetika, efektivna adresa, odredište relativnog grananja)

4. MEM: pristup podatkovnoj memoriji
   (samo memorijske instrukcije)

5. WB: upis odredišnog registarskog operanda
   (aritmetika, load)

Put podataka za računalo arhitekture MIPS

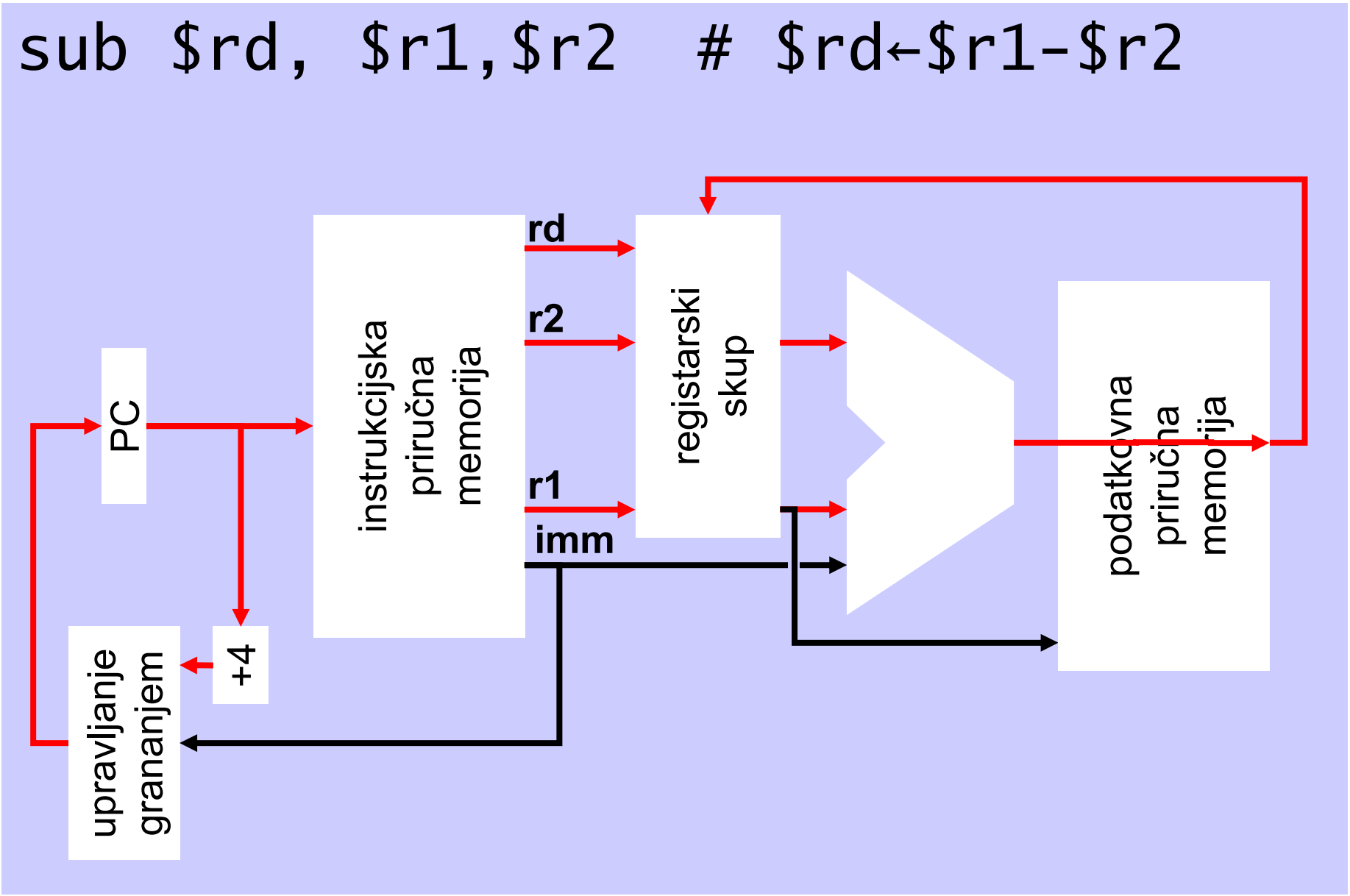


Primjer 1: registarska aritmetička instrukcija

```
sub $rd, $r1,$r2    #$rd←$r1-$r2
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati op. kod, pribaviti $r1 i $r2
- EX: oduzeti pribavljene podatke
- MEM: ništa (nije memorijska instrukcija)
- WB: upisati rezultat zbrajanja u $rd

```
sub $rd, $r1,$r2  # $rd←$r1-$r2
```

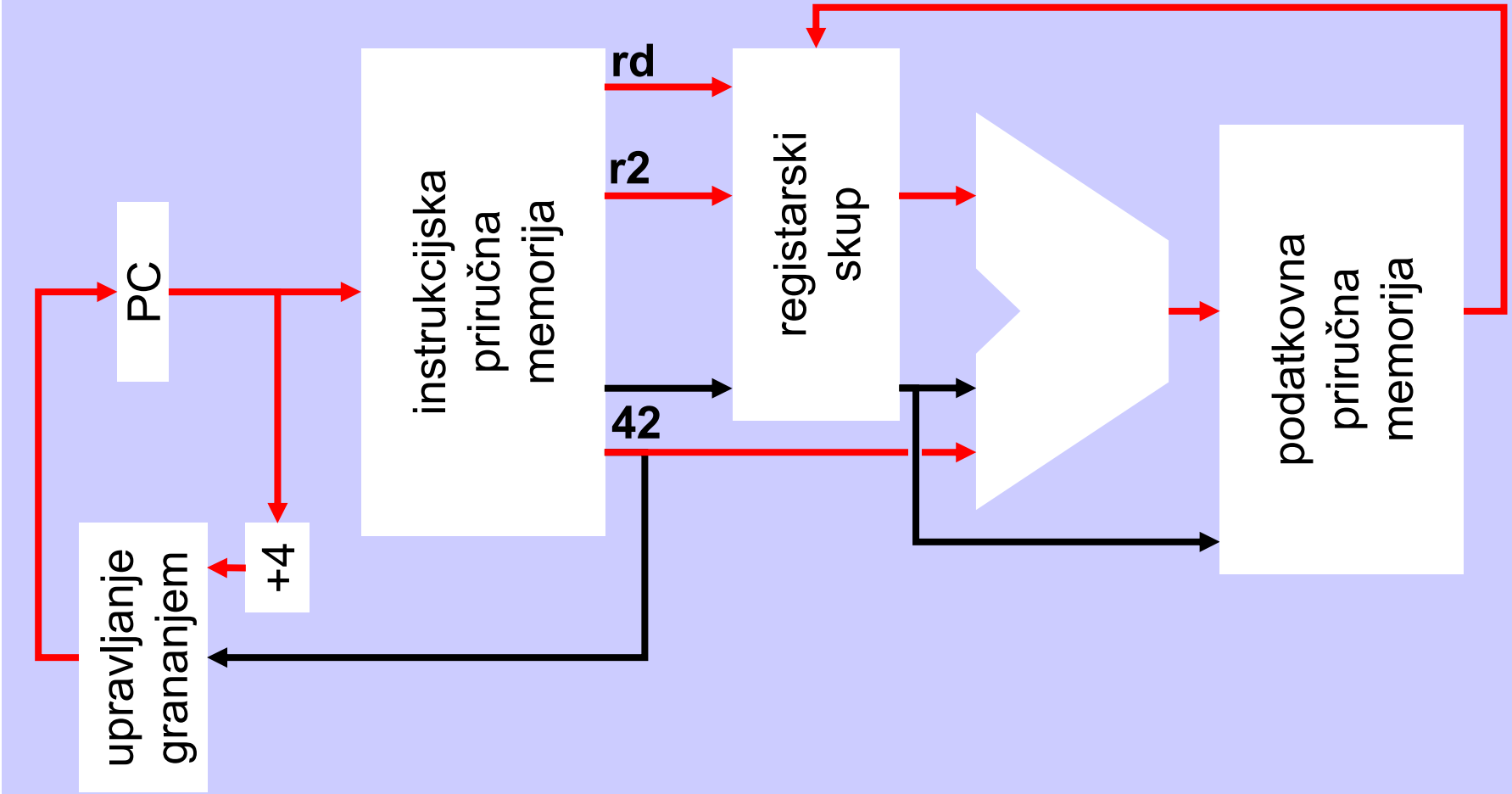


Primjer 2: memorijska instrukcija

```
lw $rd, 42($r2)    # $rd←MEM($r2+42)
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati, pribaviti `$r2`, proslijediti `42`
- EX: zbrojiti `$r2` i `42`
- MEM: učitati podatak (32 bit!) s adrese dobivene zbrajanjem
- WB: upisati učitani podatak u `$rd`

```
lw $rd, 42($r2) # $rd ← MEM[$r2+42]
```

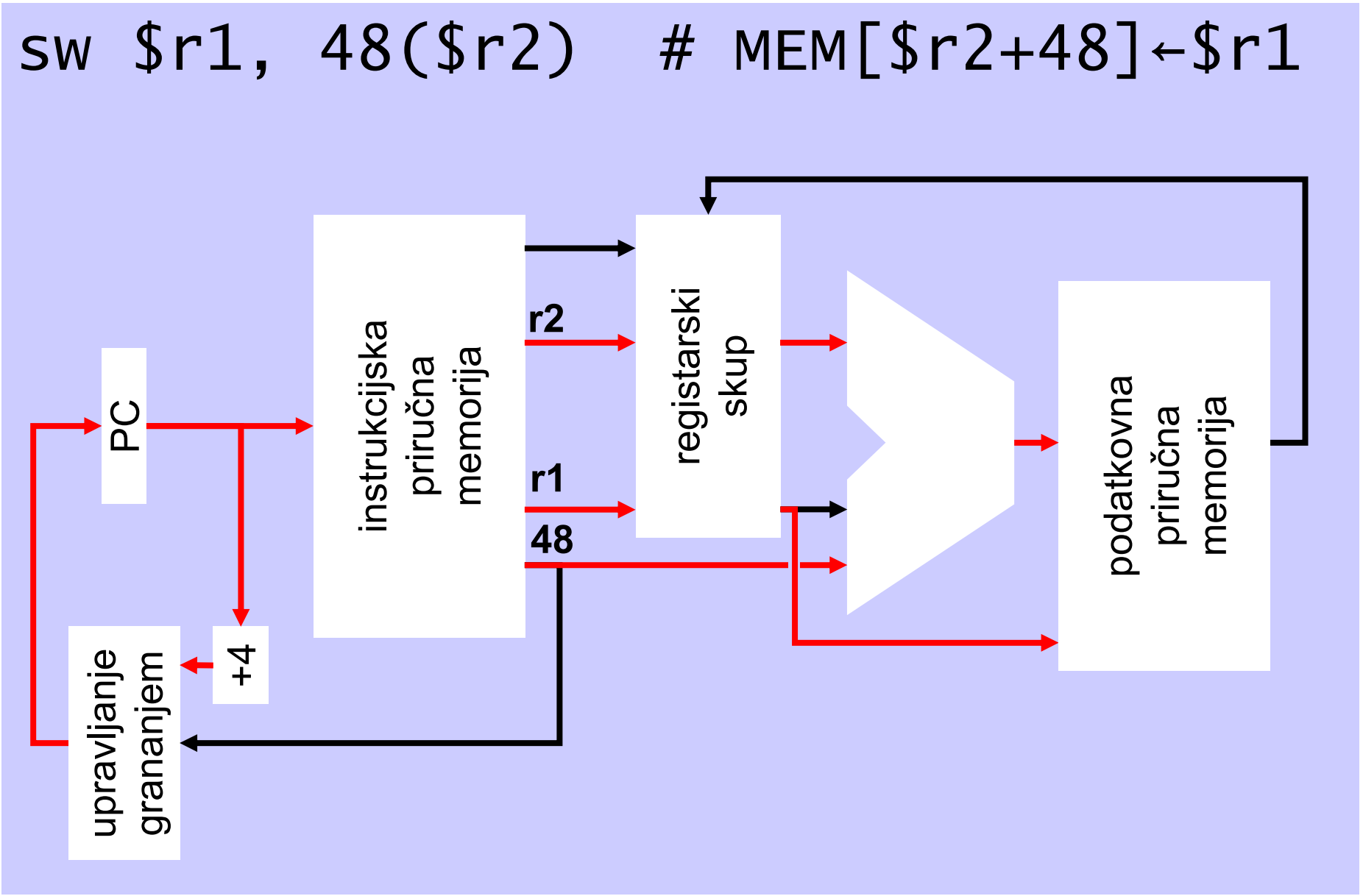


Primjer 3: memorijska instrukcija

```
sw $r1,48($r2)    # MEM($r2+48)←$r1
```

- IF: pribaviti instrukciju, PC+=4
- ID: dekodirati, pribaviti `$r2`, proslijediti `48`
- EX: zbrojiti `$r2` i `48`
- MEM: upisati podatak (32 bit!) iz registra `$r1` na adresu dobivenu zbrajanjem
- WB: ništa

```
sw $r1, 48($r2)  # MEM[$r2+48]←$r1
```



Instrukcije grananja:

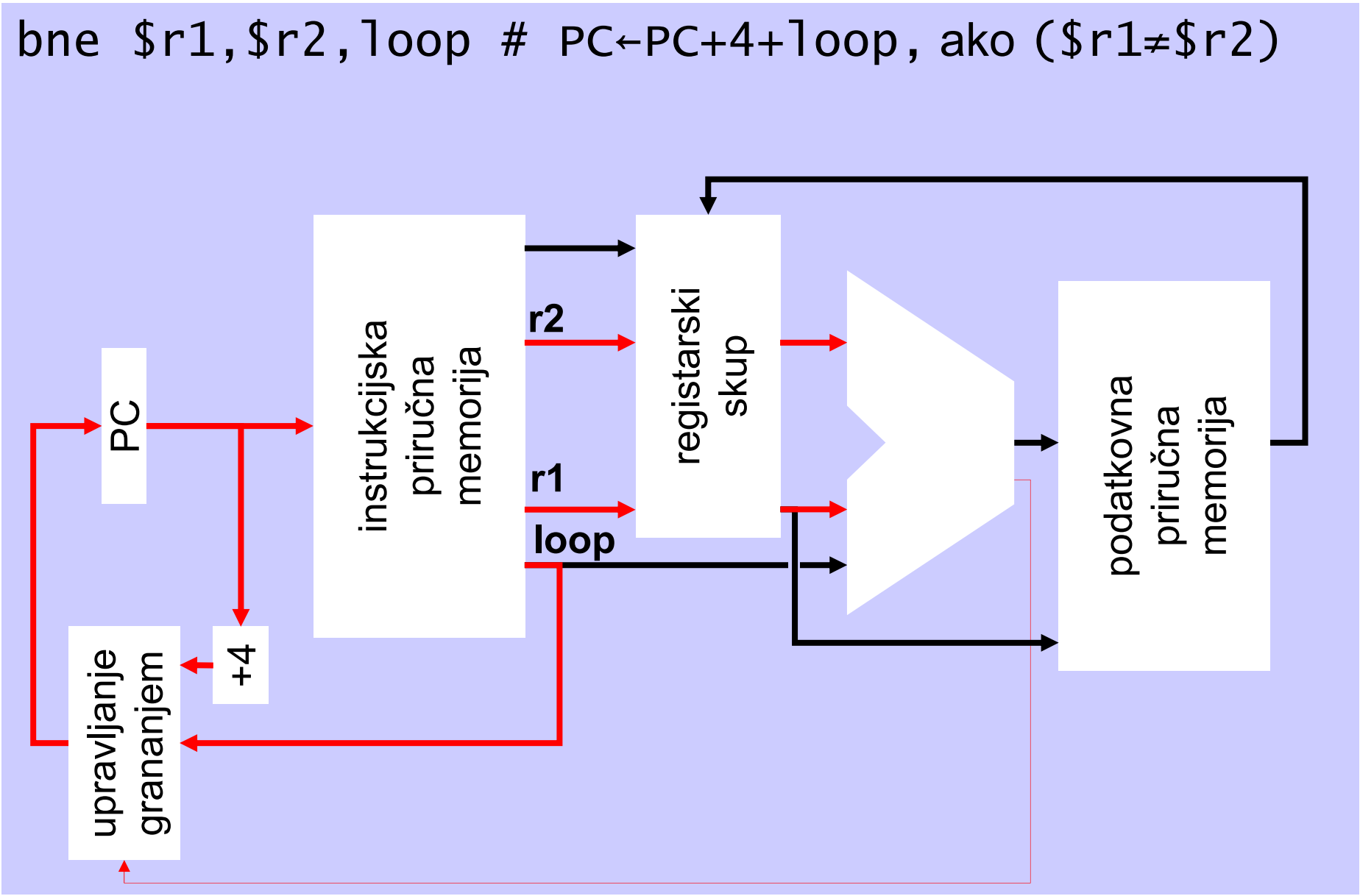
```
Npr: bne $r1,$r2,loop
          # PC←PC+4+loop, ako ($r1≠$r2)
```

- IF: pribaviti instrukciju, odrediti PC+4
- ID: dekodirati, pribaviti `$r1`,`$r2`
- EX: usporediti `$r1`,`$r2`, **zbrojiti** `PC+4` i `loop`
- MEM: ništa
- WB: upisati novi PC

Komentari:

- sklop za grananje treba posebnu ALU
- latencija ovakvog grananja će otežati protočnu izvedbu

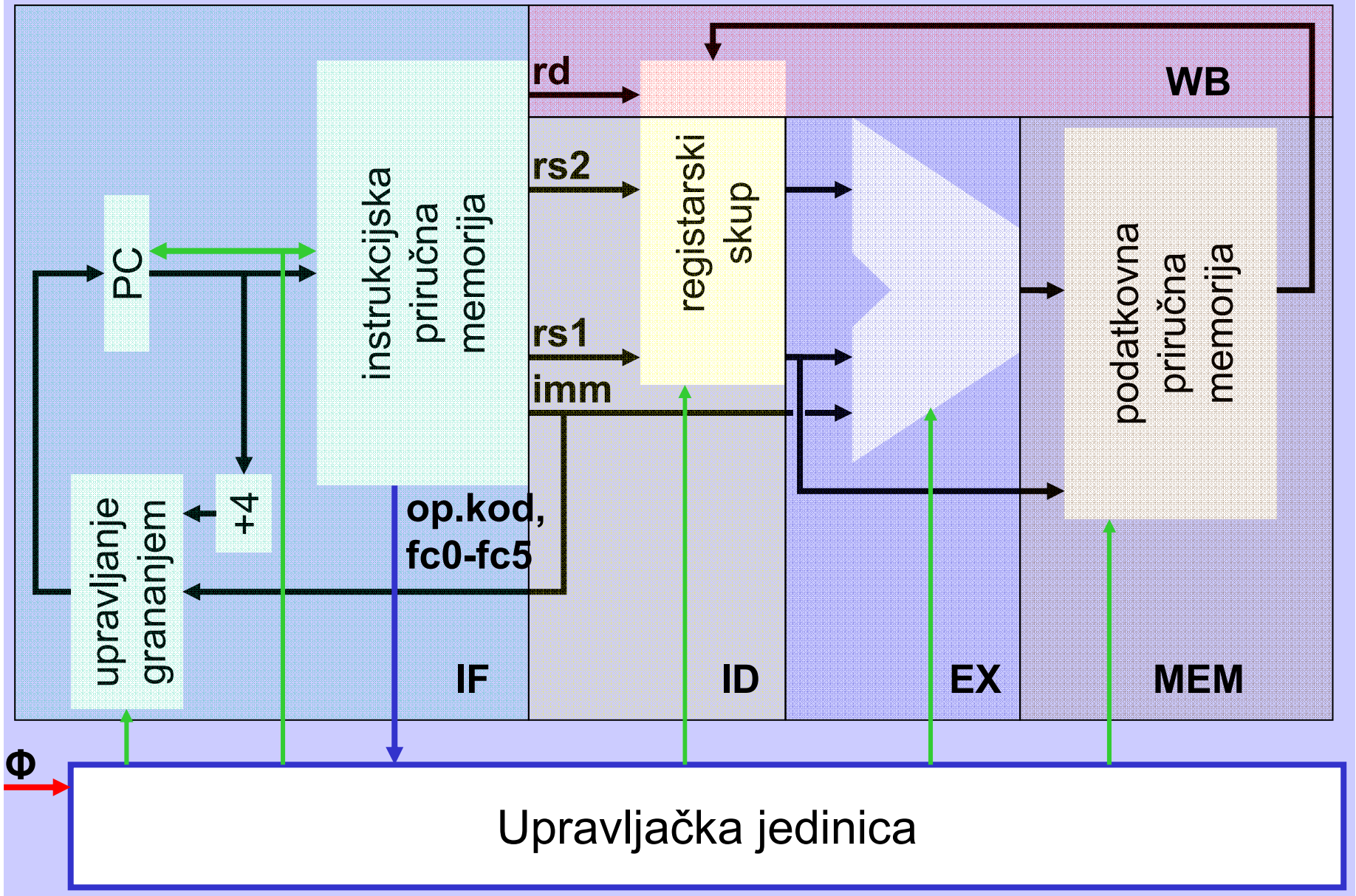
bne $r1,$r2,loop # PC←PC+4+loop, ako ($r1≠$r2)



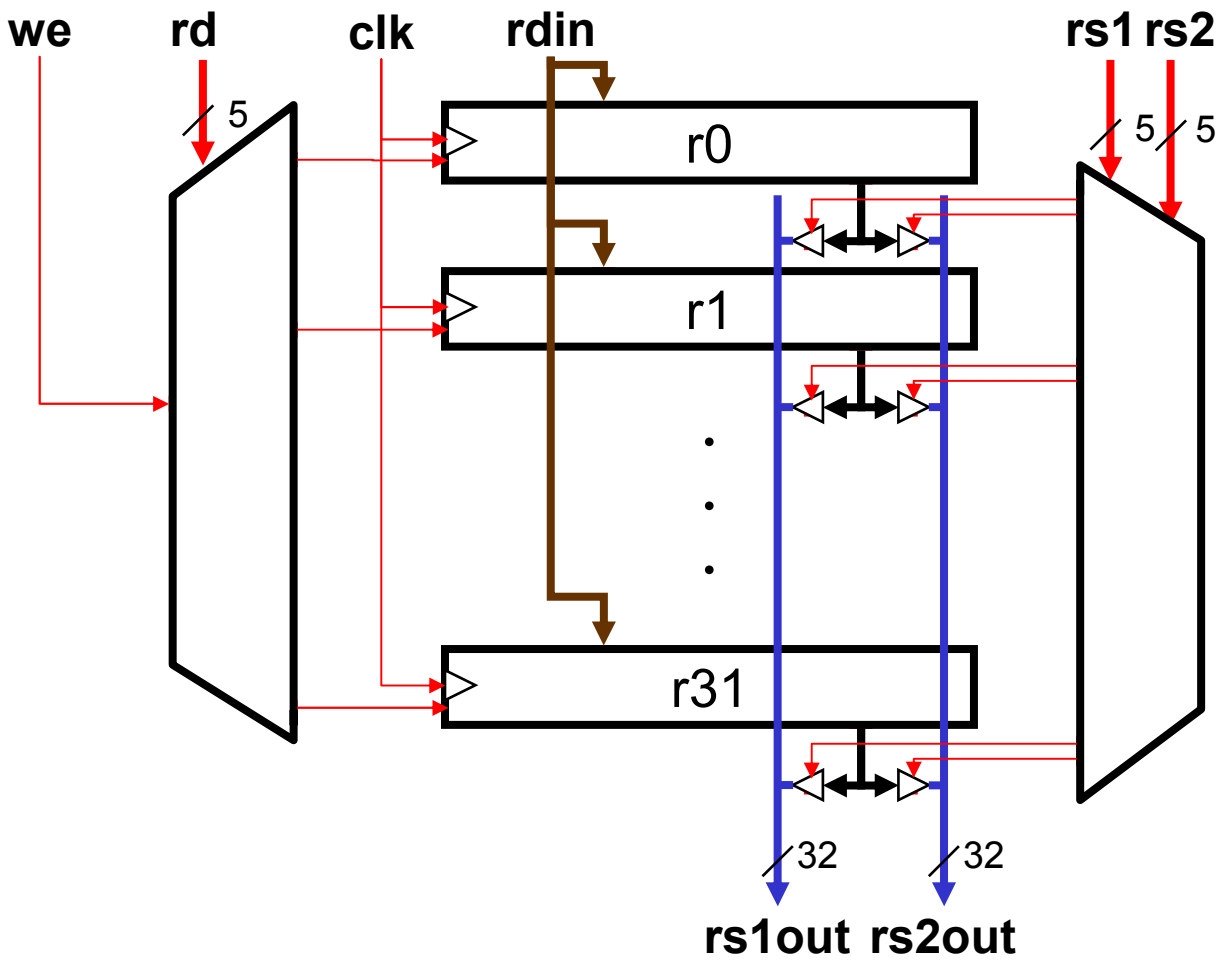
Svojstva puta podataka arhitekture MIPS:

- 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)
  - priručne memorije (detaljnije u trećem ciklusu)
- pravilan protok podataka osigurava jednostavno i efikasno upravljanje!

Arhitektura CPU: put podataka+upravljanje



Izvedba skupa registara opće namjene



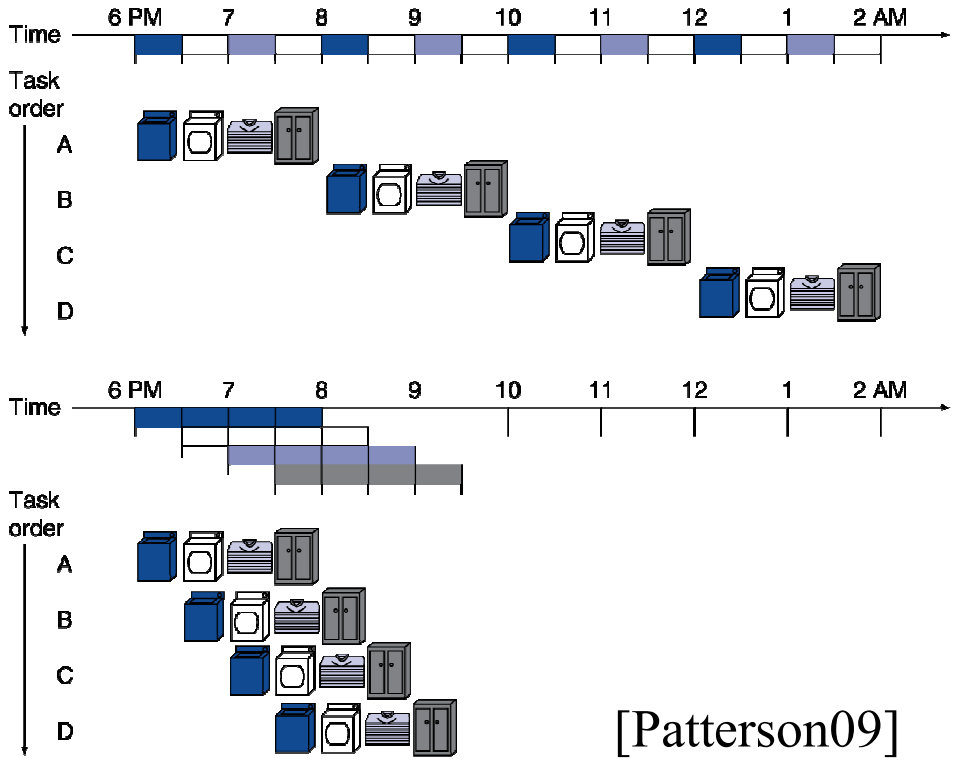
Pogled na performansu

- 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 ↑ ×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



[Patterson09]

- u slučaju četiri posla:

  ubrzanje = 8/3.5 = **2.3**

- kontinuirani rad (**n**):

  ubrzanje = 2n/(0.5n+1.5) ≈ **4**

  = broj segmenata!

- ideja posuđena iz manufaktura i proizvodnih linija [Chaplin1936]

Protočna organizacija arhitekture MIPS

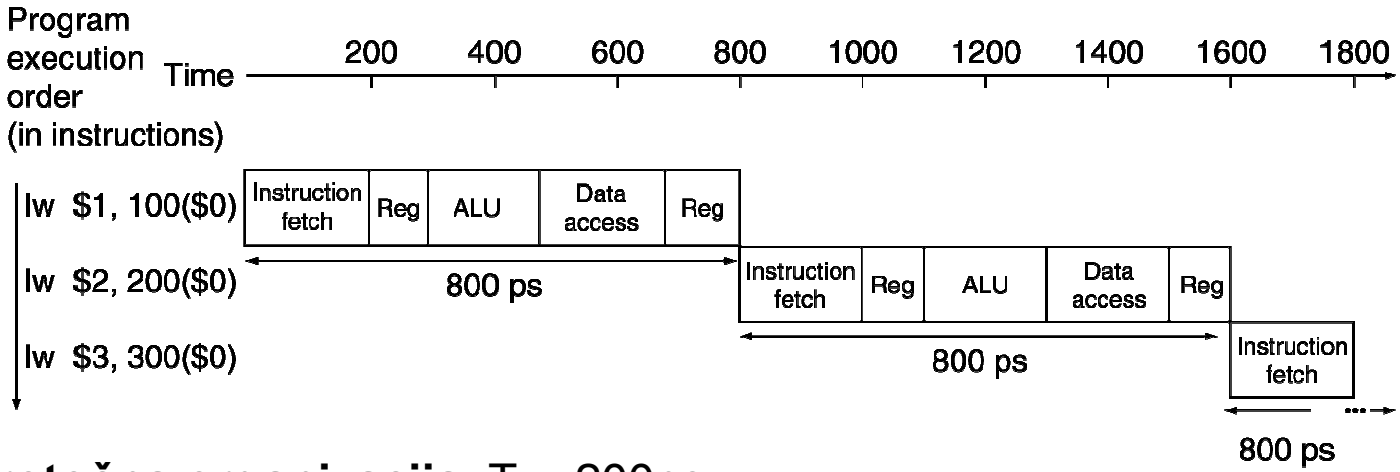
- pretpostavimo da pristup registrima traje 100ps, a ostale operacije 200ps
- tada dobivamo sljedeće latencije tipičnih instrukcija:

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

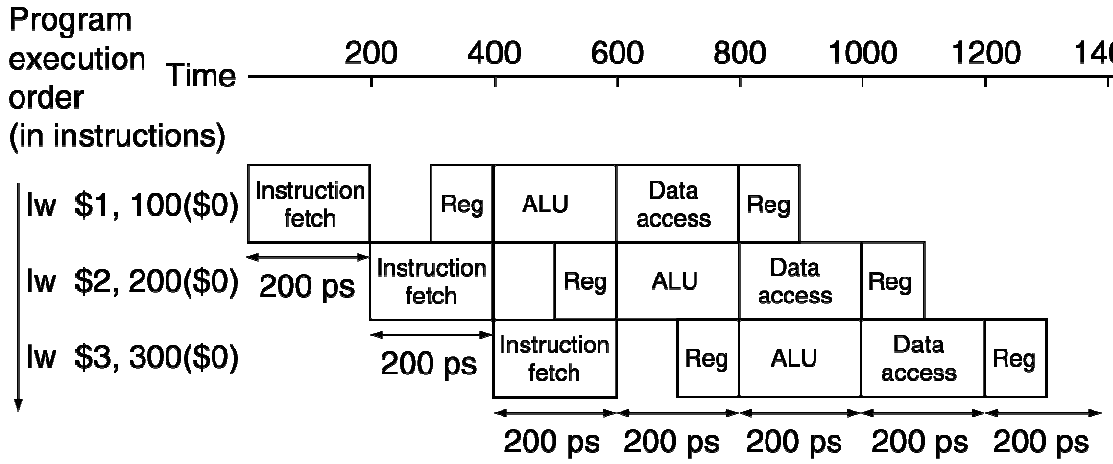
[Patterson09]

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

osnovna organizacija: T_c= 800ps



protočna organizacija: T_c= 200ps



[Patterson09]

U **prvoj** aproksimaciji, **za veći broj instrukcija** dobili bismo ubrzanje **×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, 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 ⇒ ubrzanje najviše četverostruko
- inače, ubrzanje je manje
  - u našem **školskom slučaju**, ubrzanje ovisi o učestalosti 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
  - **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

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

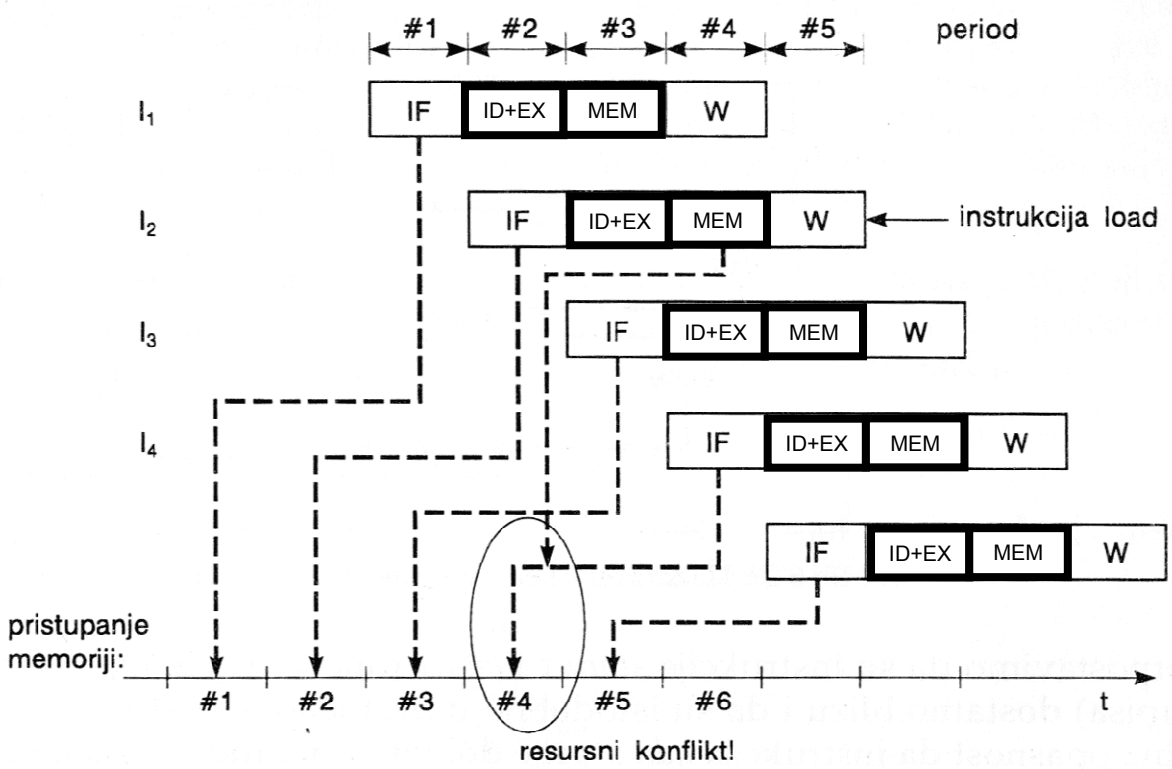
Tri razreda hazarda:

- strukturni hazard
- podatkovni hazard
- upravljački hazard

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

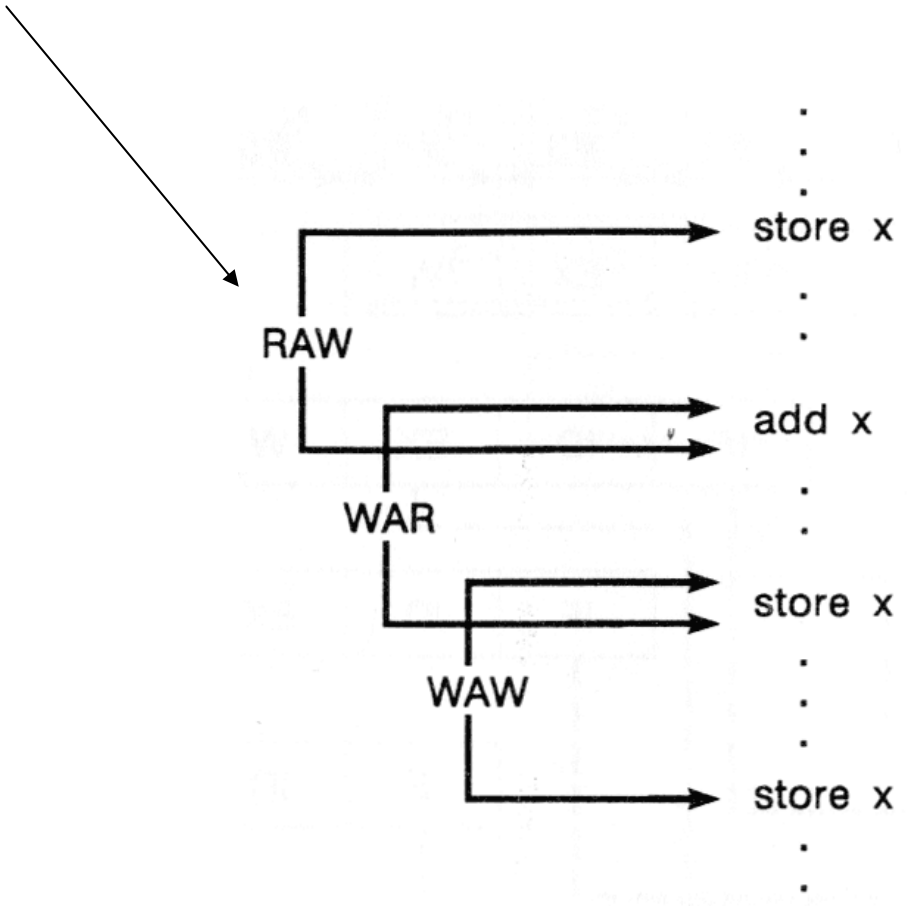
- Podatkovni hazard nastupa zbog *međuovisnosti podataka*

- 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 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:

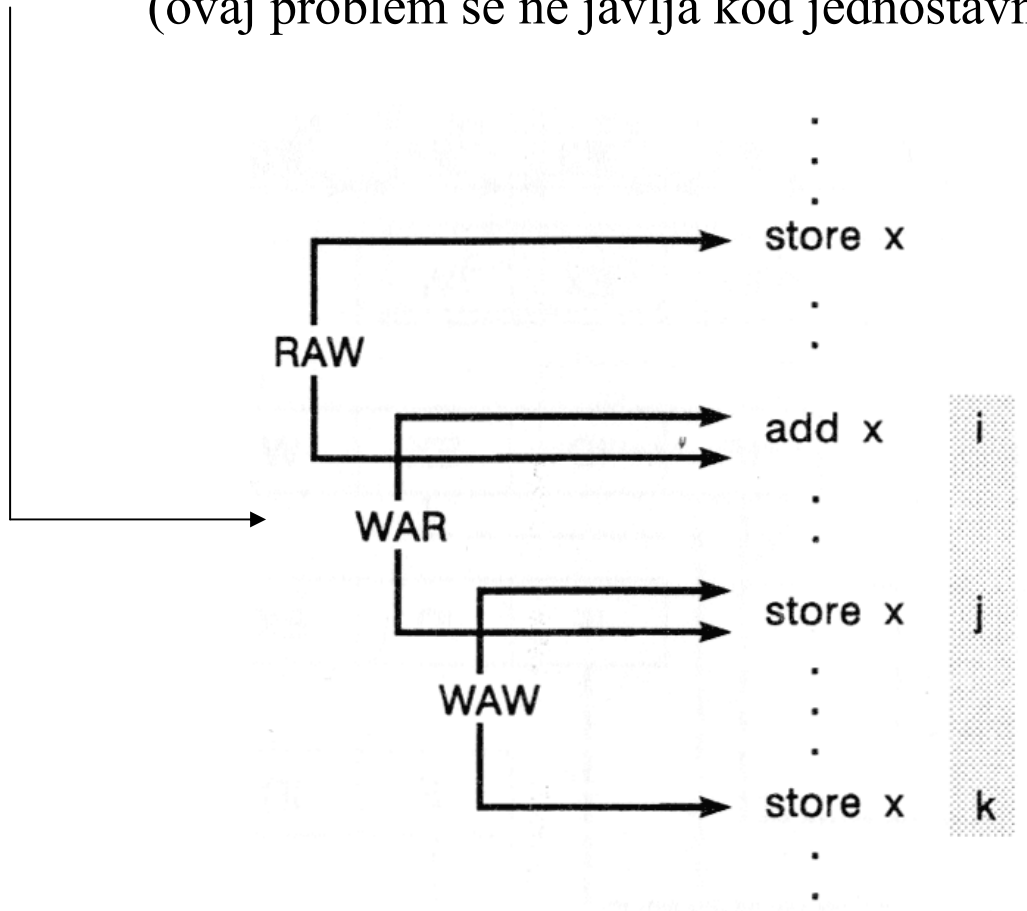
```
add  $r1, $r2,$r3

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 protočnih arhitektura)



registarski primjer:

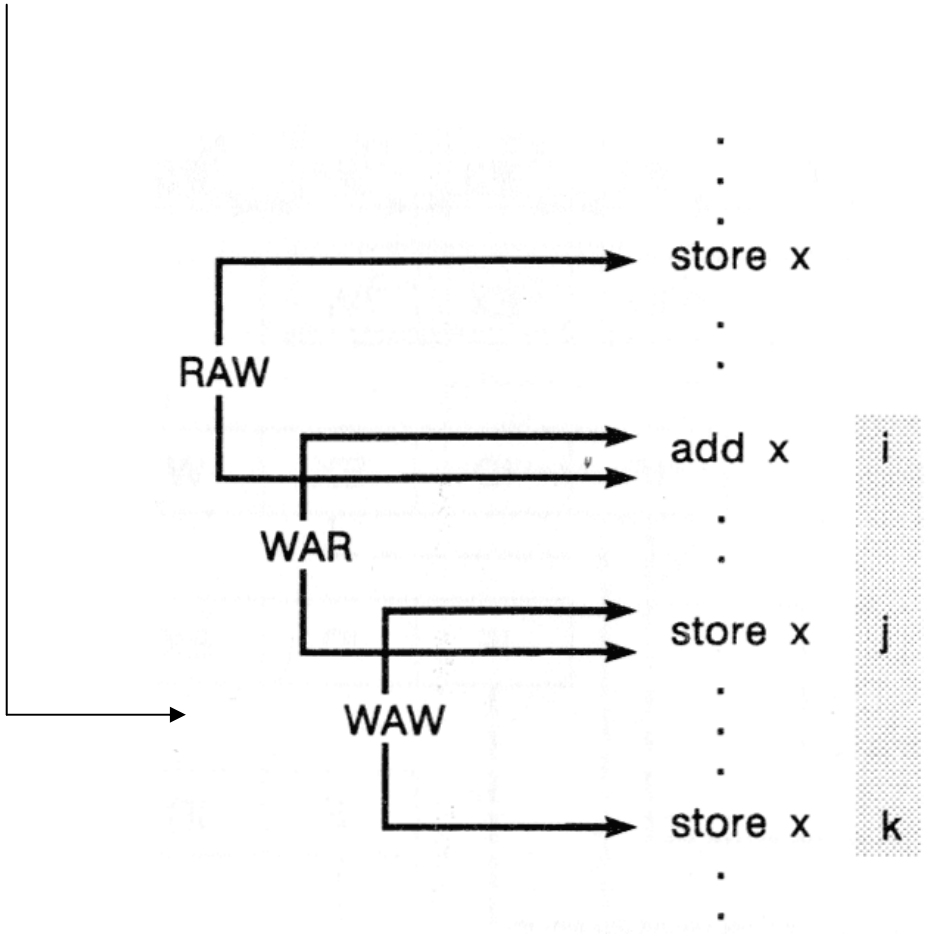
```
add  $r1, $r2,$r3

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 memorijskojl lokaciji *x*
(problem nastaje ako se instrukcija *j* izvede poslije instrukcije *k,*
ovaj problem se ne javlja kod jednostavnih protočnih arhitektura*)*



registarski primjer:

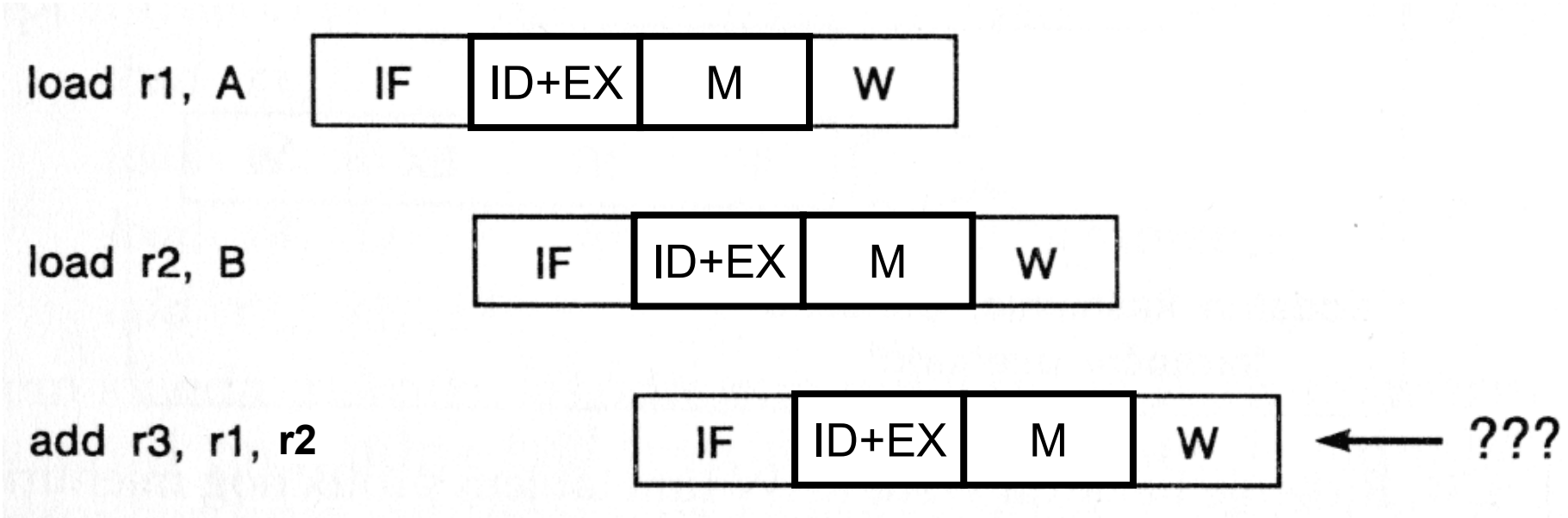
```
add  $r1,$r2,$r3

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*



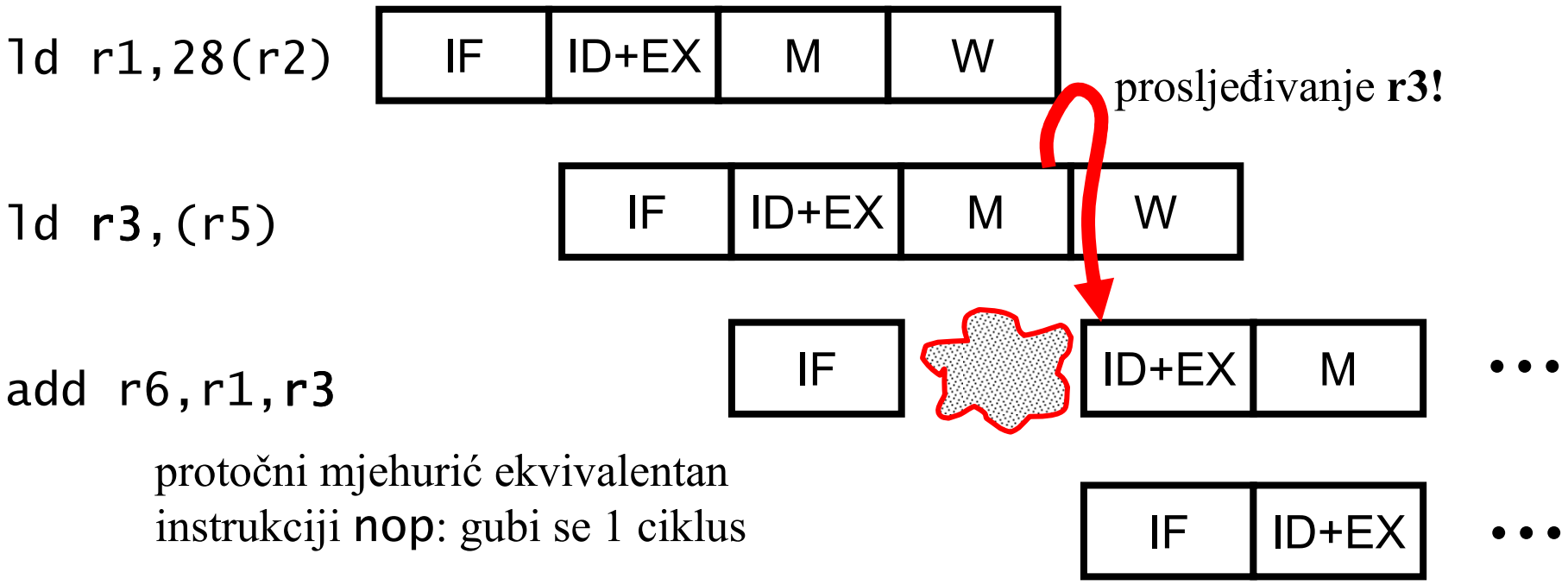
Saniranje hazarda RAW nakon instrukcije load:

- 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*

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]→ID+EX[i+1]:

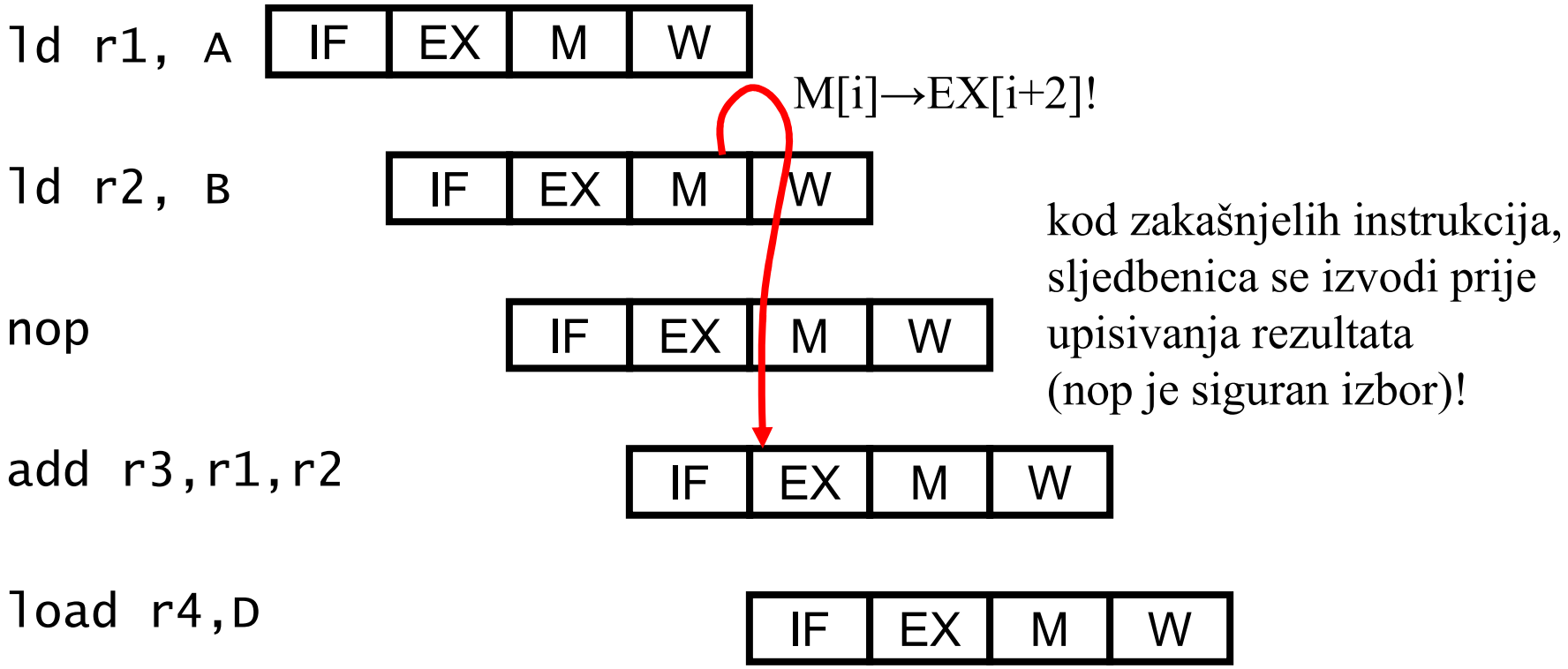


protočni mjehurić ekvivalentan instrukciji nop: gubi se 1 ciklus

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**

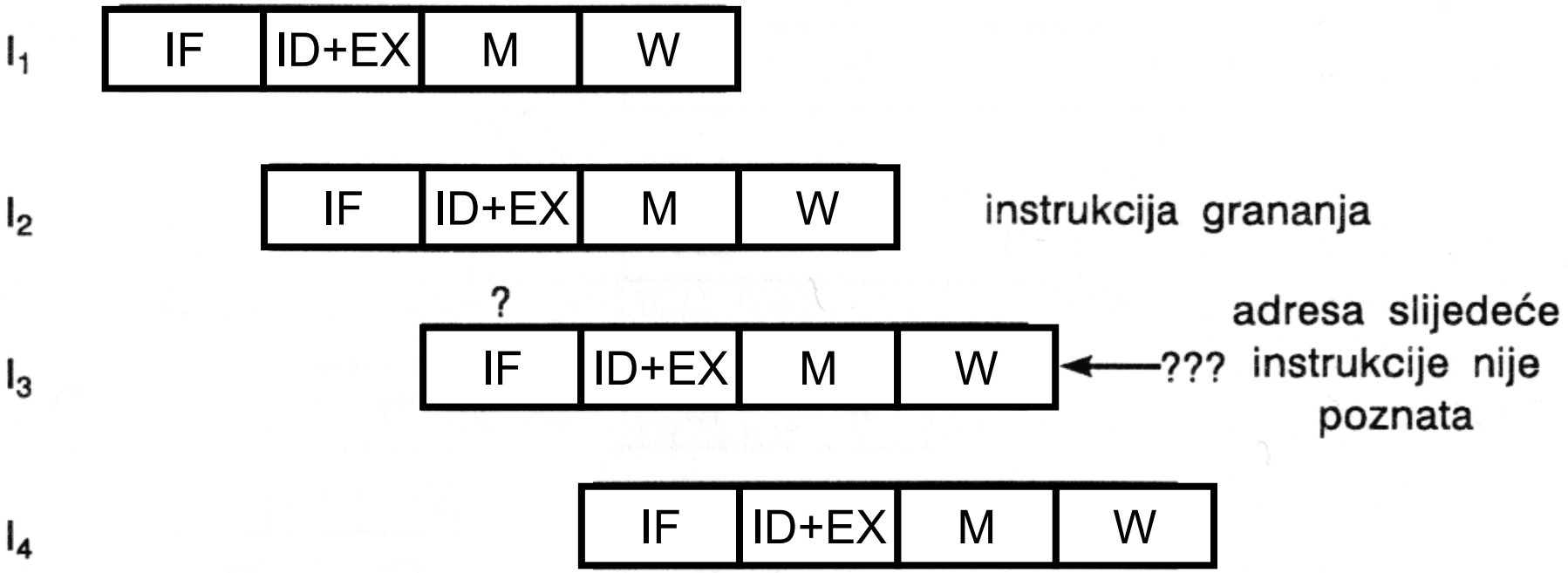
Primjer (zakašnjelo čitanje):

C := A + B; E:= D

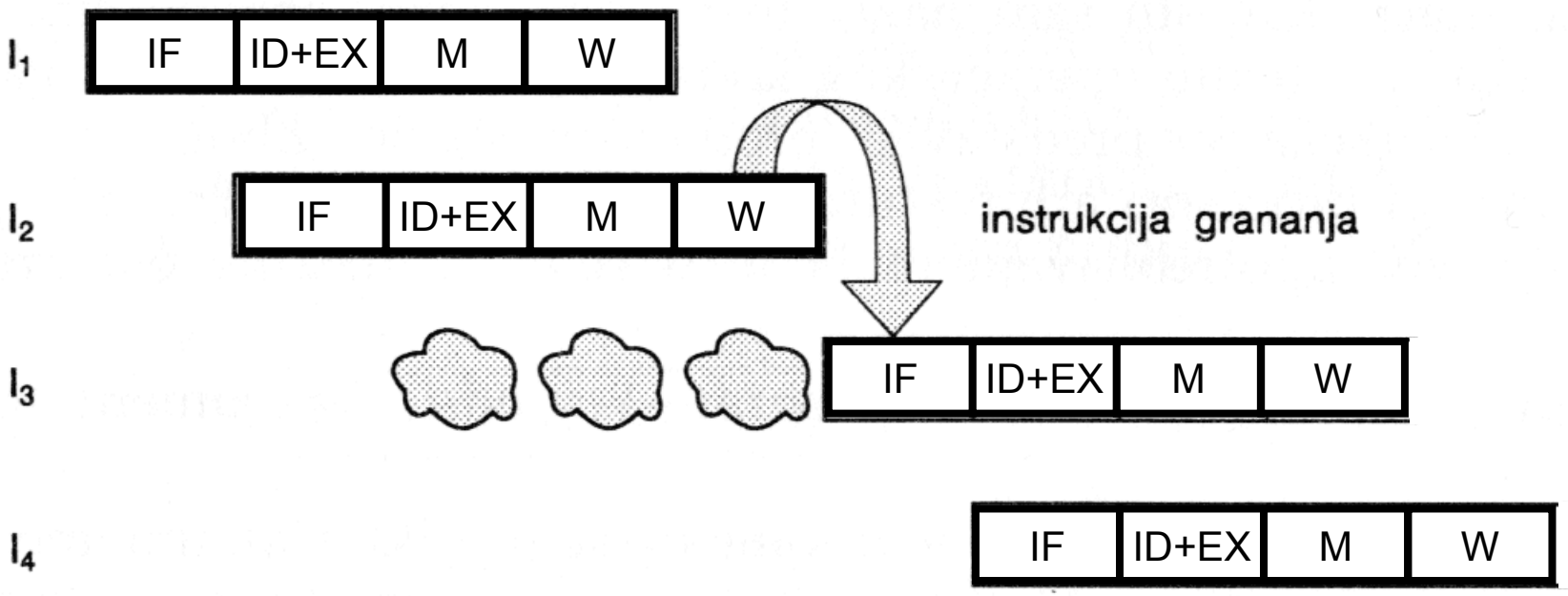
Upravljački hazard

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] → 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`

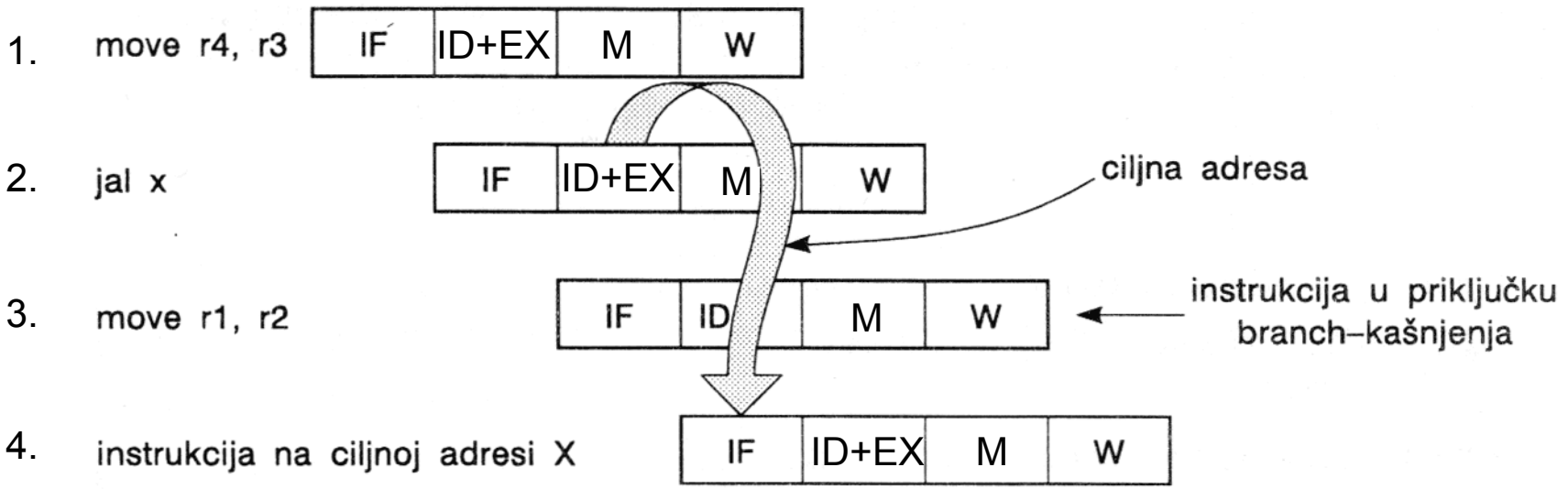
Kašnjenje samo s jednim mjehurićem!

Mjehurići se mogu potpuno zaobići **zakašnjelim grananjem**

- u tom slučaju koristimo prosljeđivanje ID[i] → IF[i+2]

Zakašnjelo grananje:

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



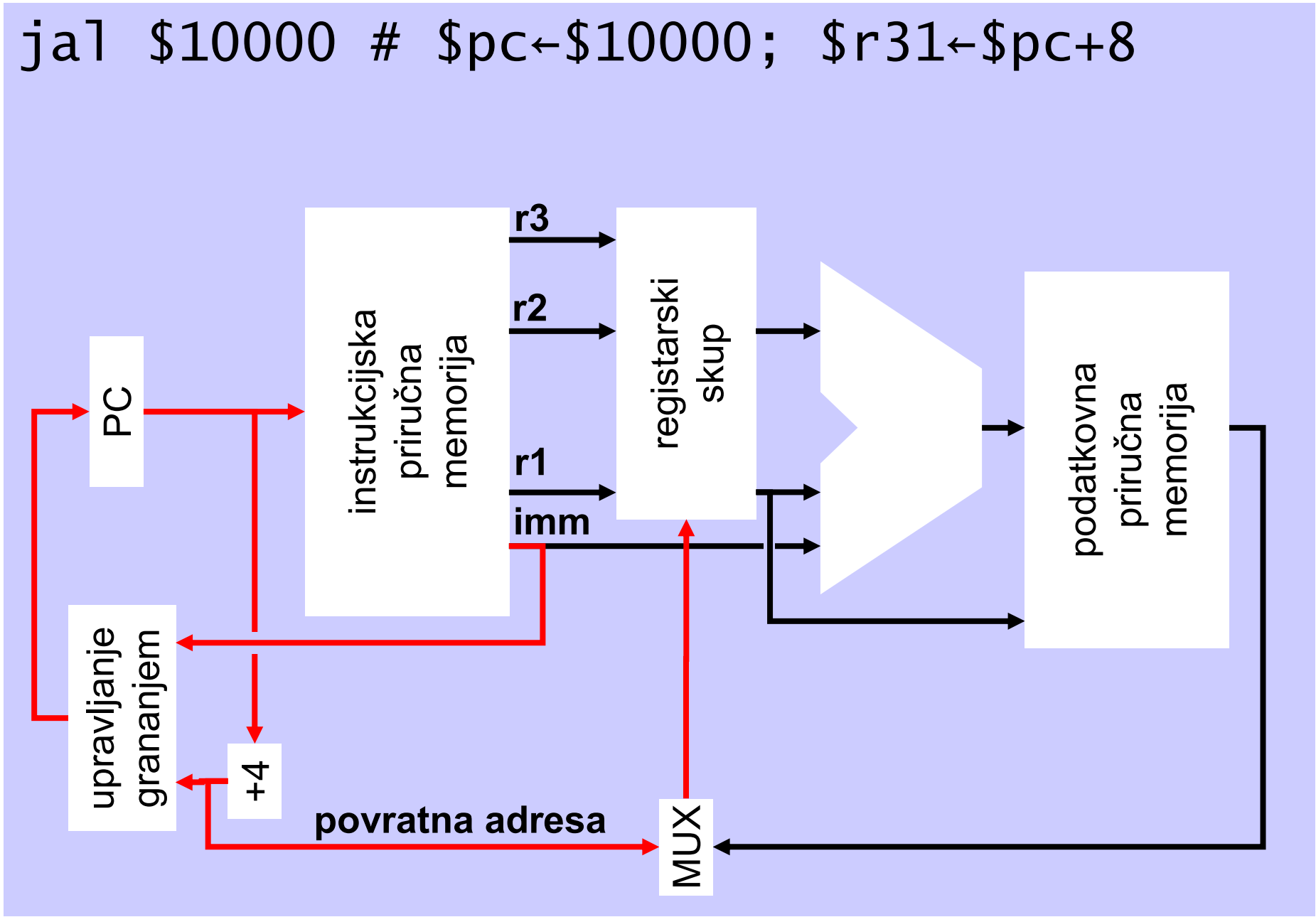
Primjer za računalo MIPS: instrukcija grananja

```
jal $10000 # $pc←$10000; $r31←$pc+8
```

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

Pažnja: koristi se **zakašnjelo** grananje, instrukcija neposredno nakon instrukcije grananja se također izvodi!

```
jal $10000 # $pc←$10000; $r31←$pc+8
```



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
  - μoperacije po segmentima jednako traju

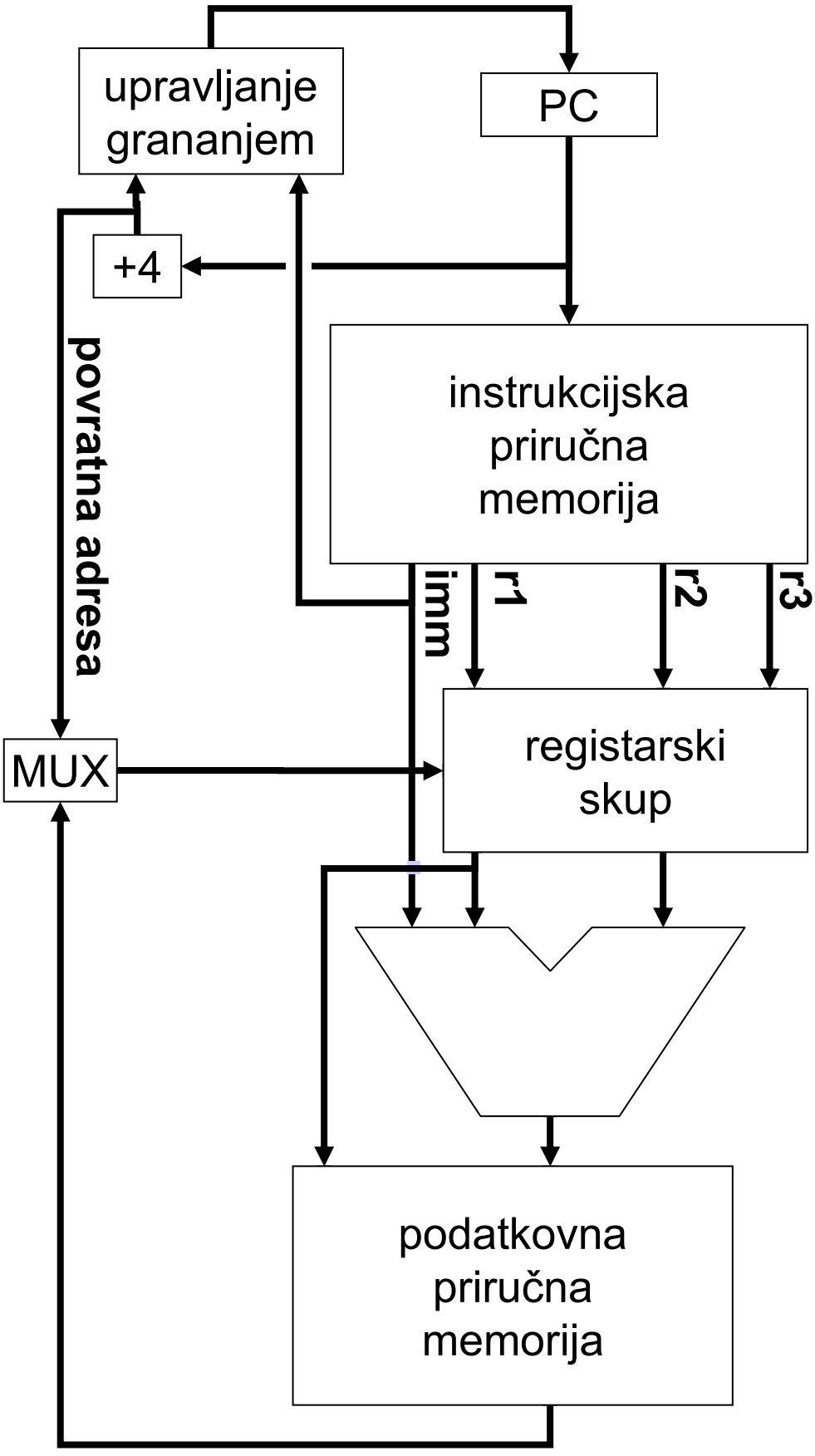
Protočnost, sažetak (2):

- protočnost omogućava efikasno iskorištavanje instrukcijskog paralelizma
- faktor ubrzanja ×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.


upravljanje grananjem
PC
+4
povratna adresa
instrukcijska priručna memorija
imm
r1
r2
r3
MUX
registarski skup
podatkovna priručna memorija