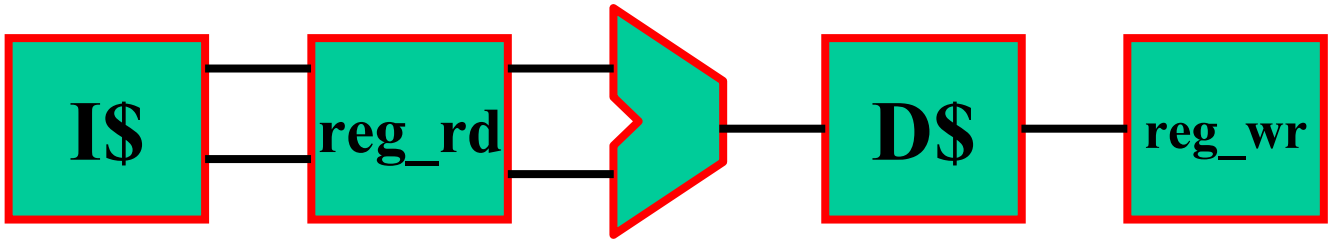
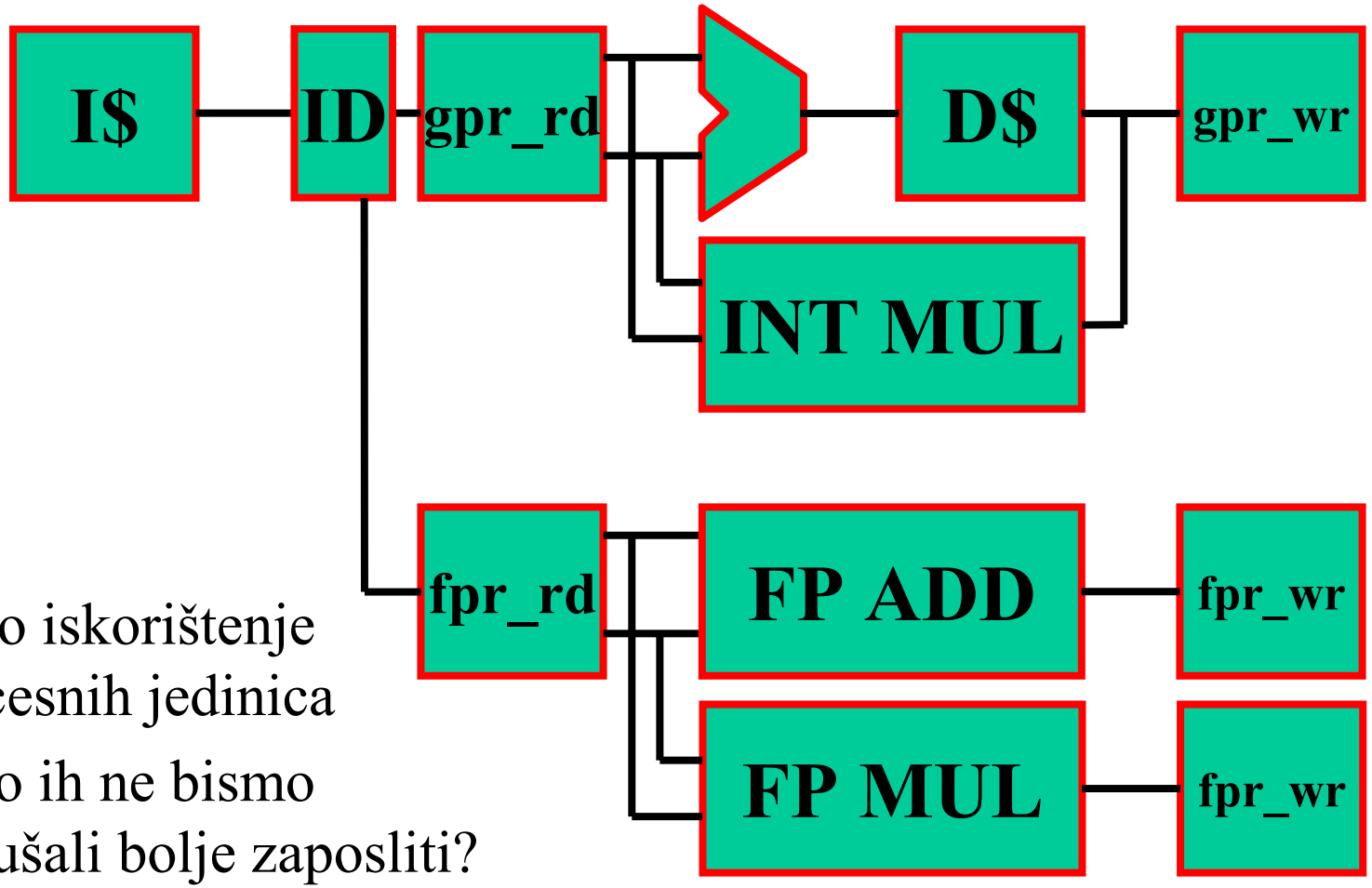
# Od protočnosti do višestrukog izdavanja




- 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 uklopiti ih u temeljni cjevovod

**Složena skalarna protočna organizacija:**

- segmenti različite dužine
- veće mogućnosti bez velikog usporenja temeljnih operacija



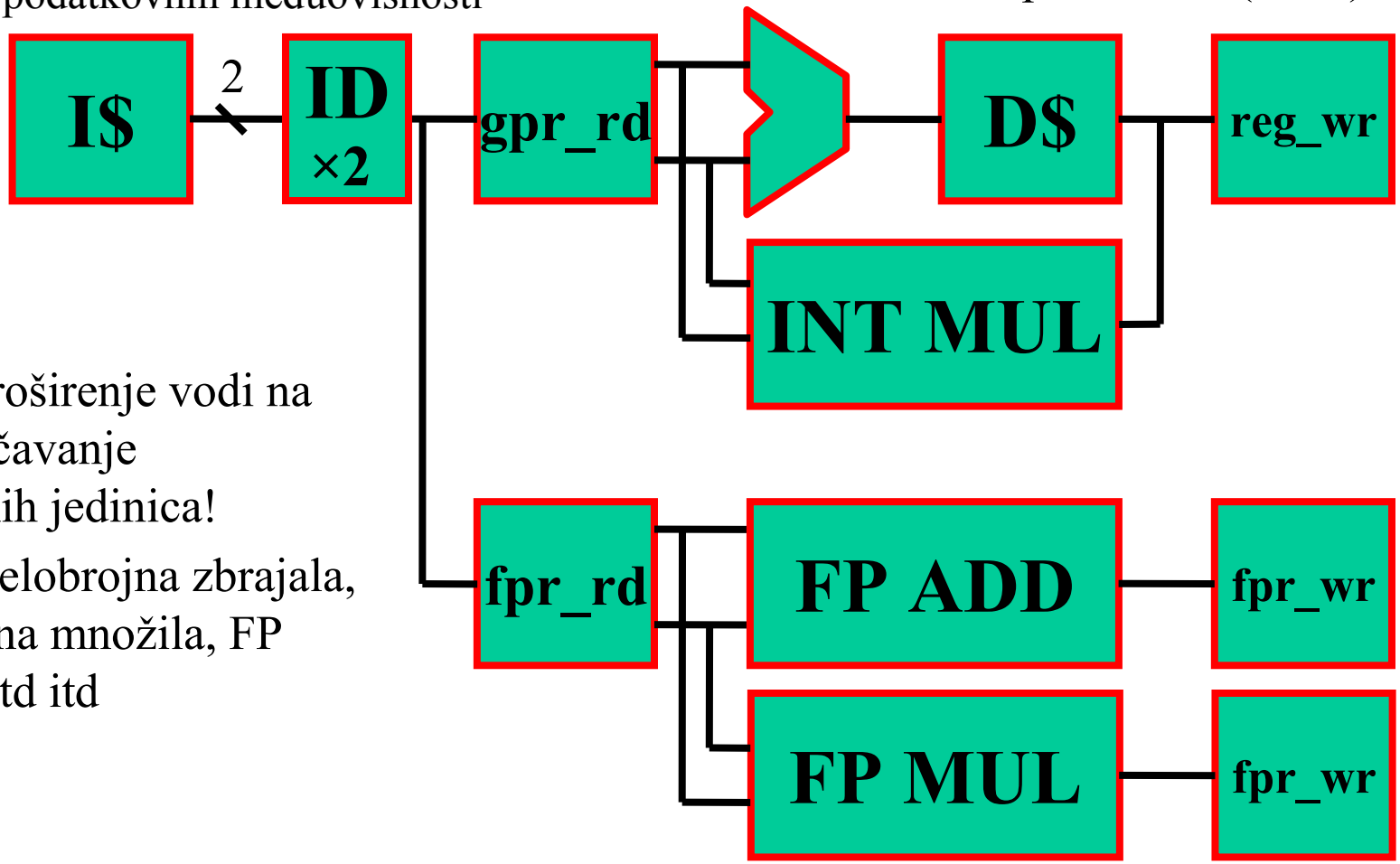

- slabo iskorištenje procesnih jedinica
- zašto ih ne bismo pokušali bolje zaposliti?

## 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 međuovisnosti

Predstavnici:

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




- daljnje proširenje vodi na udvostručavanje funkcijskih jedinica!
- po dva cjelobrojna zbrajala, cjelobrojna množila, FP zbrajala itd itd

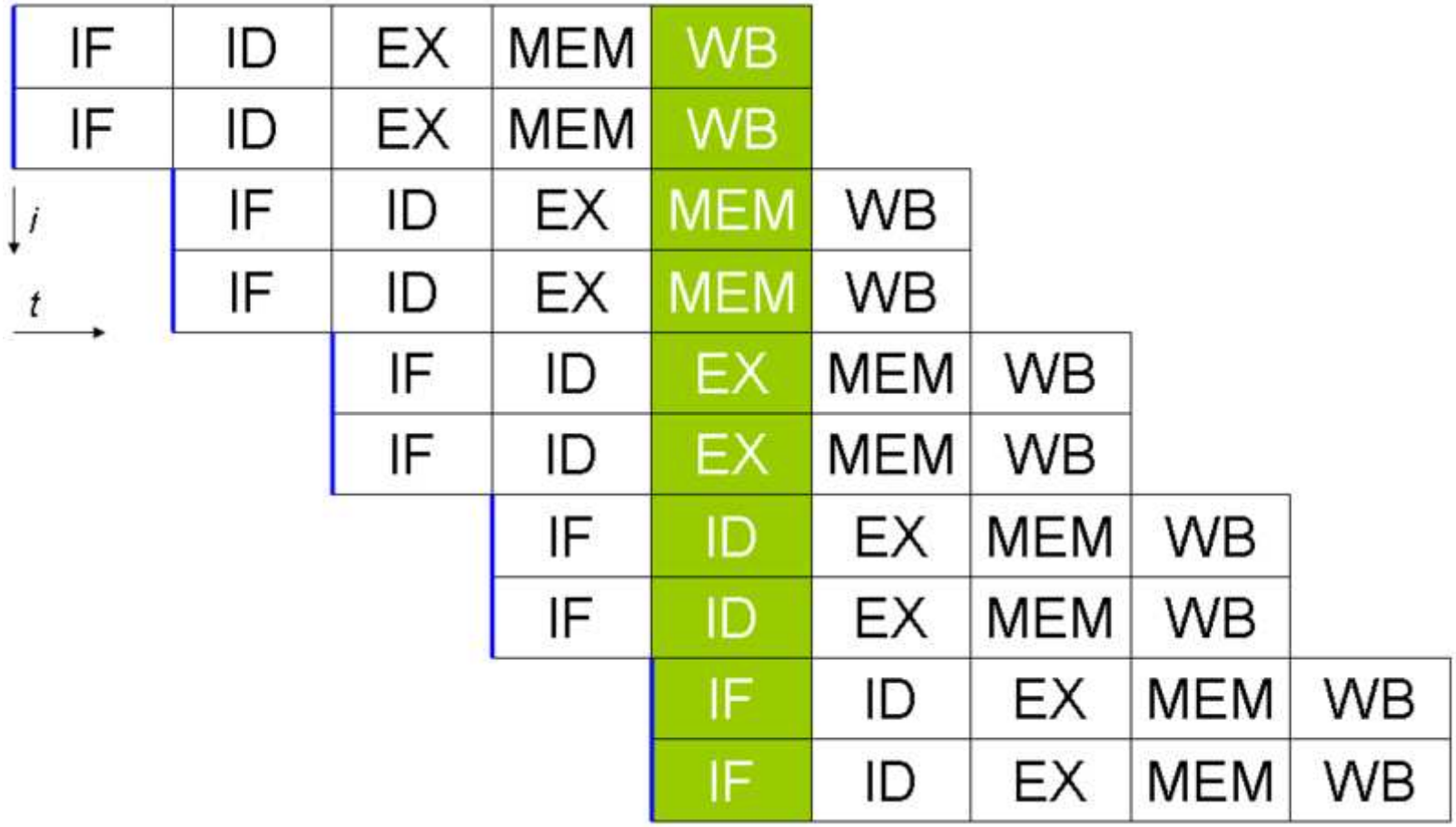
**Ideja**: istovremenim **izdavanjem** više instrukcija razotkriti više instrukcijskog paralelizma nego što to omogućava protočnost

- u implementaciji: replicirani protočni segmenti!
- 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:

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

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




[Wikipedia]

# Dva glavna pristupa višestrukom izdavanju:

**1. statičko izdavanje** (VLIW)

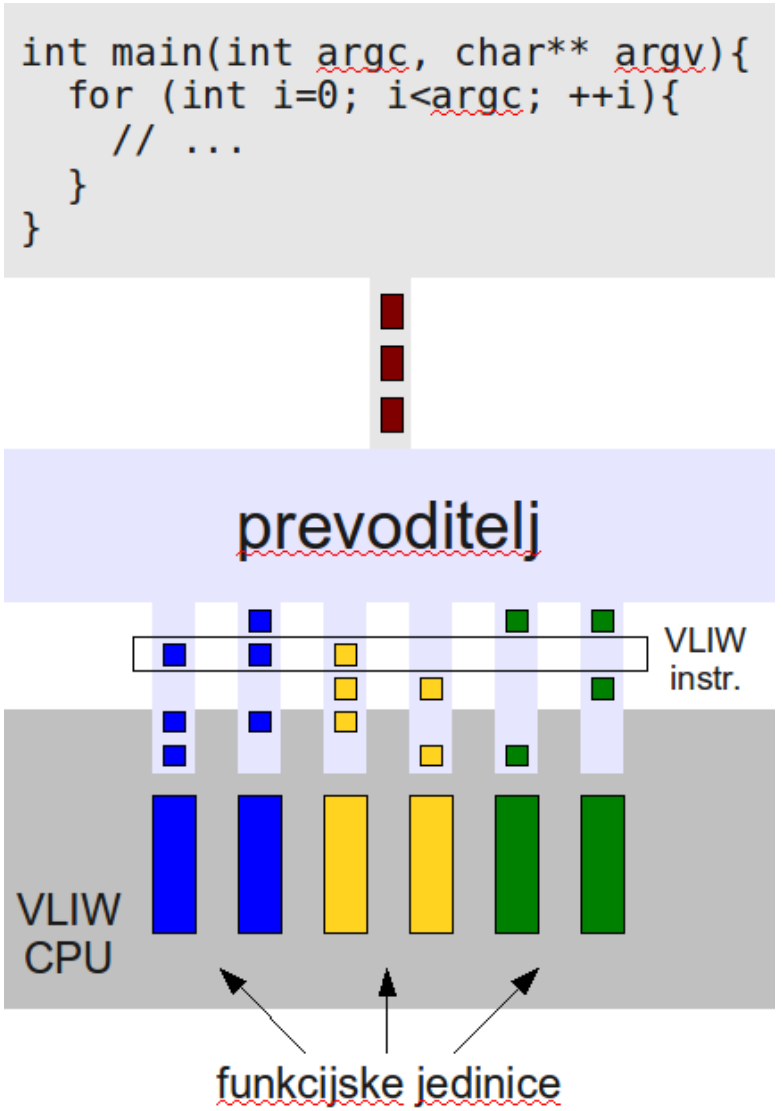
- prevoditelj grupira instrukcije koje se izdaju zajedno
- **pakete** instrukcija promatramo kao "duge instrukcije"
- prevoditelj detektira hazarde i po potrebi umeće nop-ove

**2. dinamičko izdavanje** ("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

# 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 međuovisnosti unutar paketa!
  - popuniti neiskorištene okvire (nop)



# MIPS s dvostrukim statičkim izdavanjem

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

| adresa | vrsta instrukcije | protočni segmenti | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n | ALU/branch | IF | ID | EX | MEM | WB | | |
| 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 |

[Patterson08]

# Hazardi pri dvostrukom statičkom 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 statičkog raspoređivanja (MIPS, 2×izdavanje)

```
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 # $s3 .. int* limit
```

| | 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 (slabije od vršnog CPI = 0.5) [Patterson08]

$n_I$ = 5/iteraciji

# Kako prevoditelj može pomoći:

- optimizacija 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 |

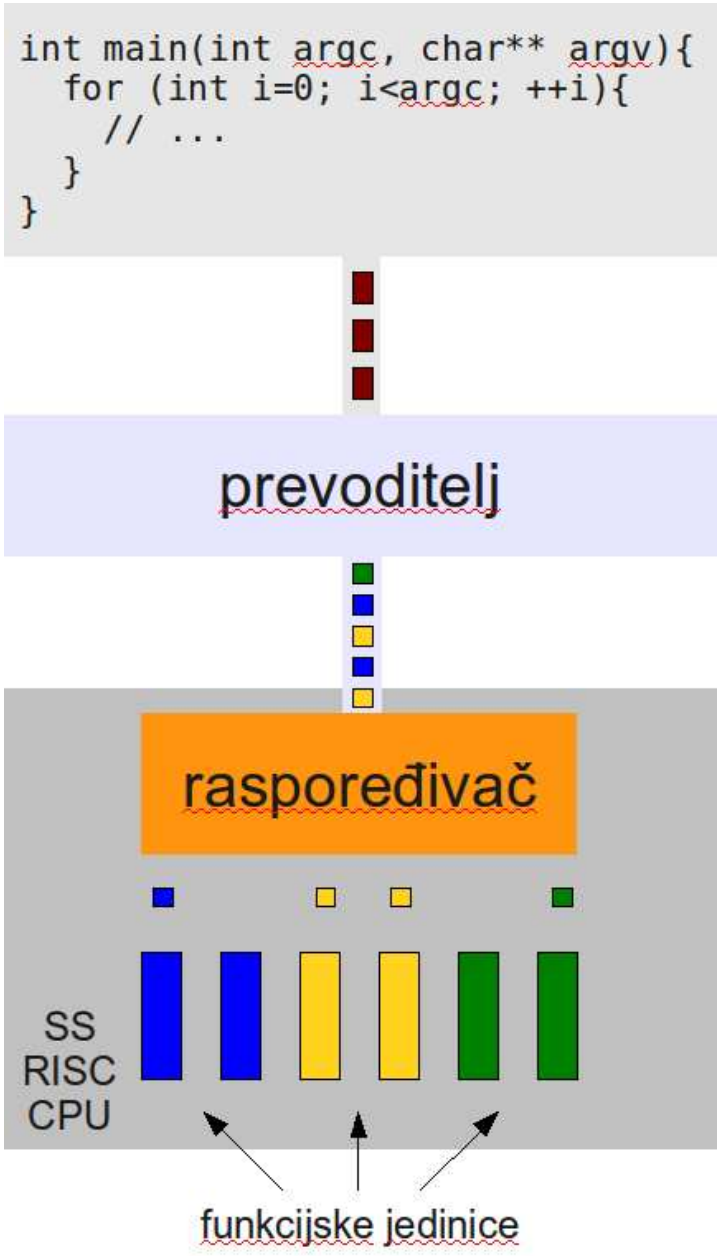
CPI = 8/14 = 0.6 (vršni CPI = 0.5) [Patterson08]

$n_I$ = 3.5/iteraciji

**Cijena**: veći potrošak registara, duži 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



# 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` (redosljed je svojevrsni strukturni hazard kod računala sa statičkim raspoređivanjem):

- 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 funkcijsku 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: redosljed sub i add!
add.d f10, f0,f10
```

- stvarna latencija instrukcije lw nije poznata u trenutku prevođenja 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`
- $\Rightarrow$ najbolje rezultate postiže **dinamičko raspoređivanje!**

# Motivacija za dinamičko raspoređivanje

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

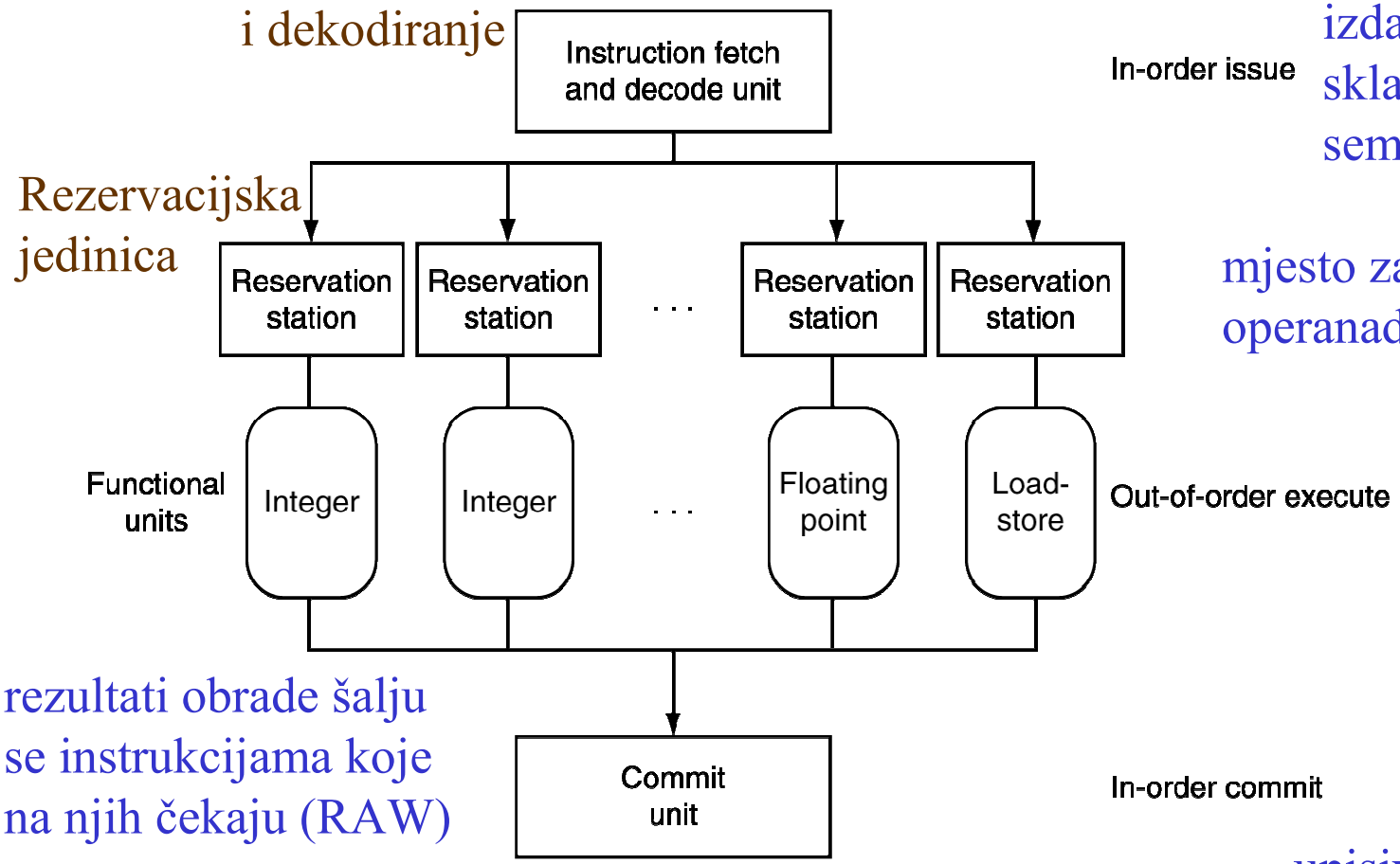
Međutim, sve hazarde nije moguće predvidjeti statičkom analizom:

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

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

# Suvremeni CPU-a s dinamičkim raspoređivanjem

Jedinica za pribavljanje
i dekodiranje




izdavanje instrukcija u skladu s redosljedom i semantikom programa

Rezervacijska
jedinica

mjesto za primanje i čuvanje operanada unstrukcije

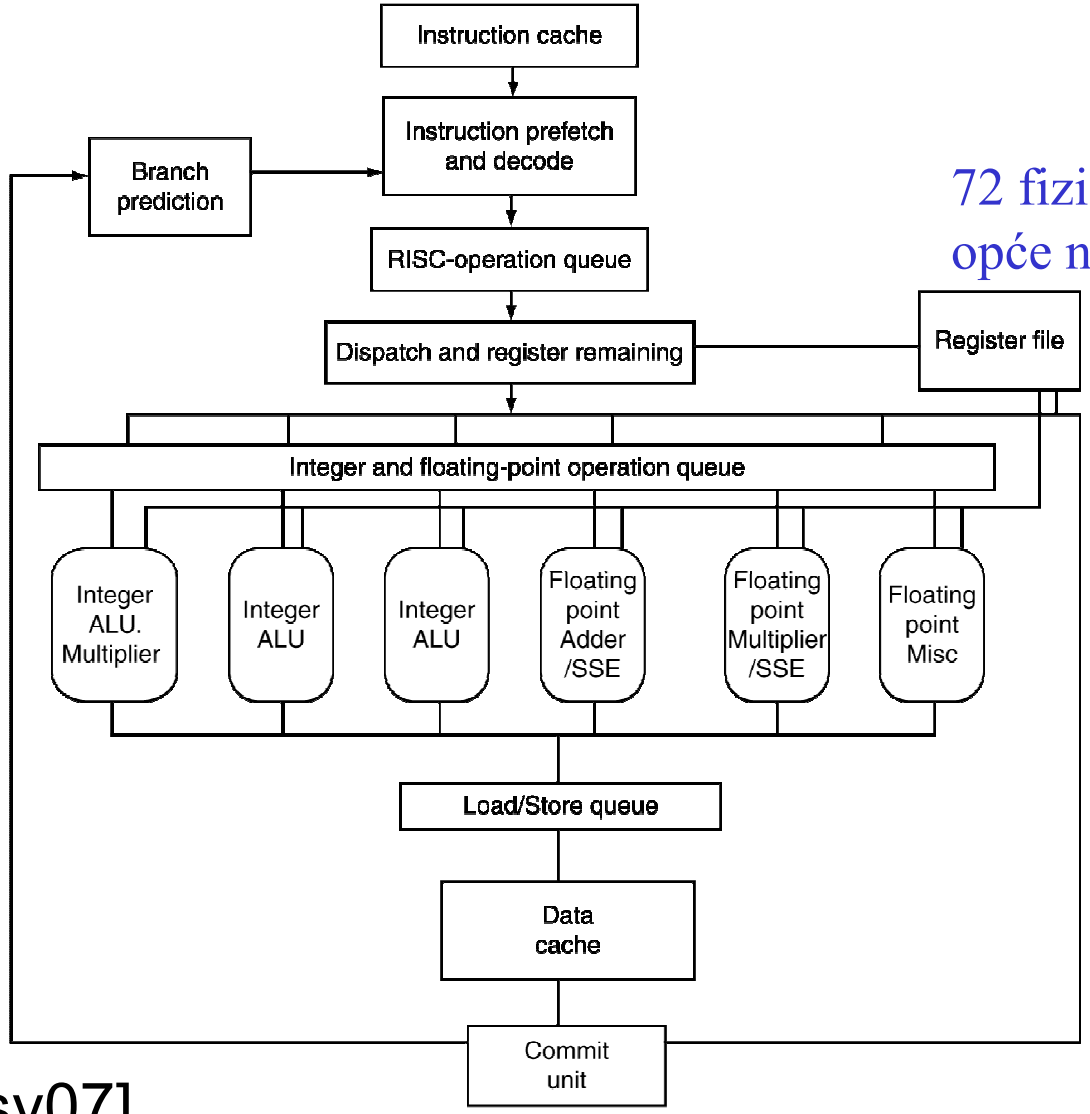
rezultati obrade šalju se instrukcijama koje na njih čekaju (RAW)

Upisna jedinica
(reorder buffer)

upisivanje u odredišne registre u skladu s originalnim redosljedom

[Hennessy07]

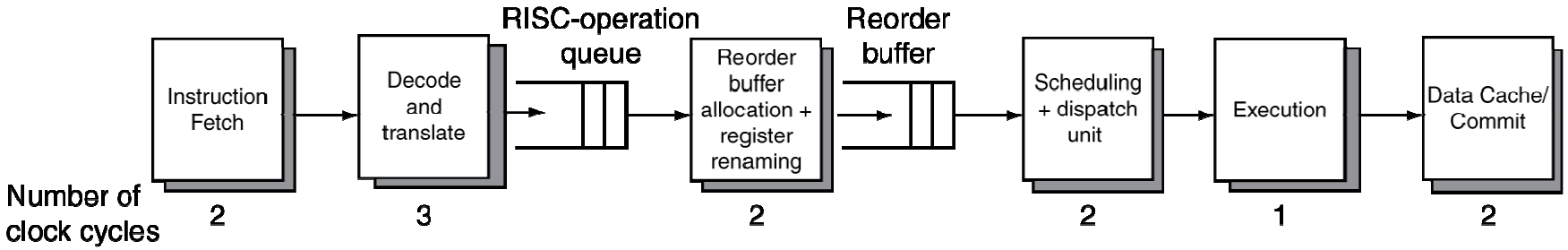
# Organizacija procesora Opteron X4




72 fizička registra opće namjene!

[Hennessy07]

# 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?

# **ILP sažetak**: SISD ulazi u zasićenje

- Agresivniji pristupi korištenja ILP-a (superskalarnost itd.) pomažu, ali njihova primjena rezultira slabijim iskorištenjem resursa
  - tranzistori (površina integriranog sklopa)
  - mogućnosti razvojnog odjela
  - energija

- 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 se to brzo promijeni...
    - 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];
}
```