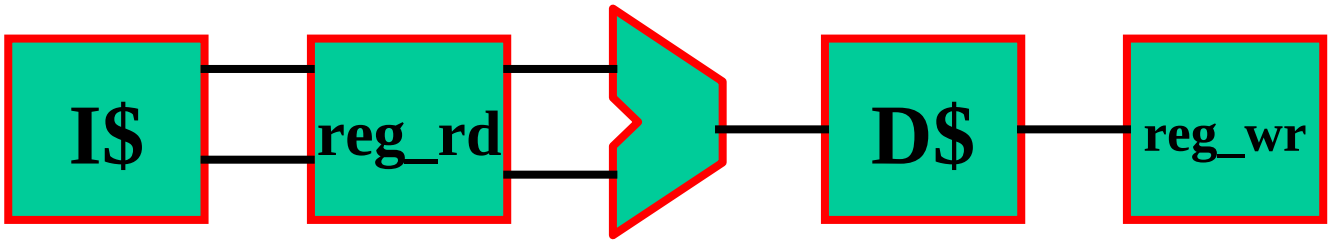
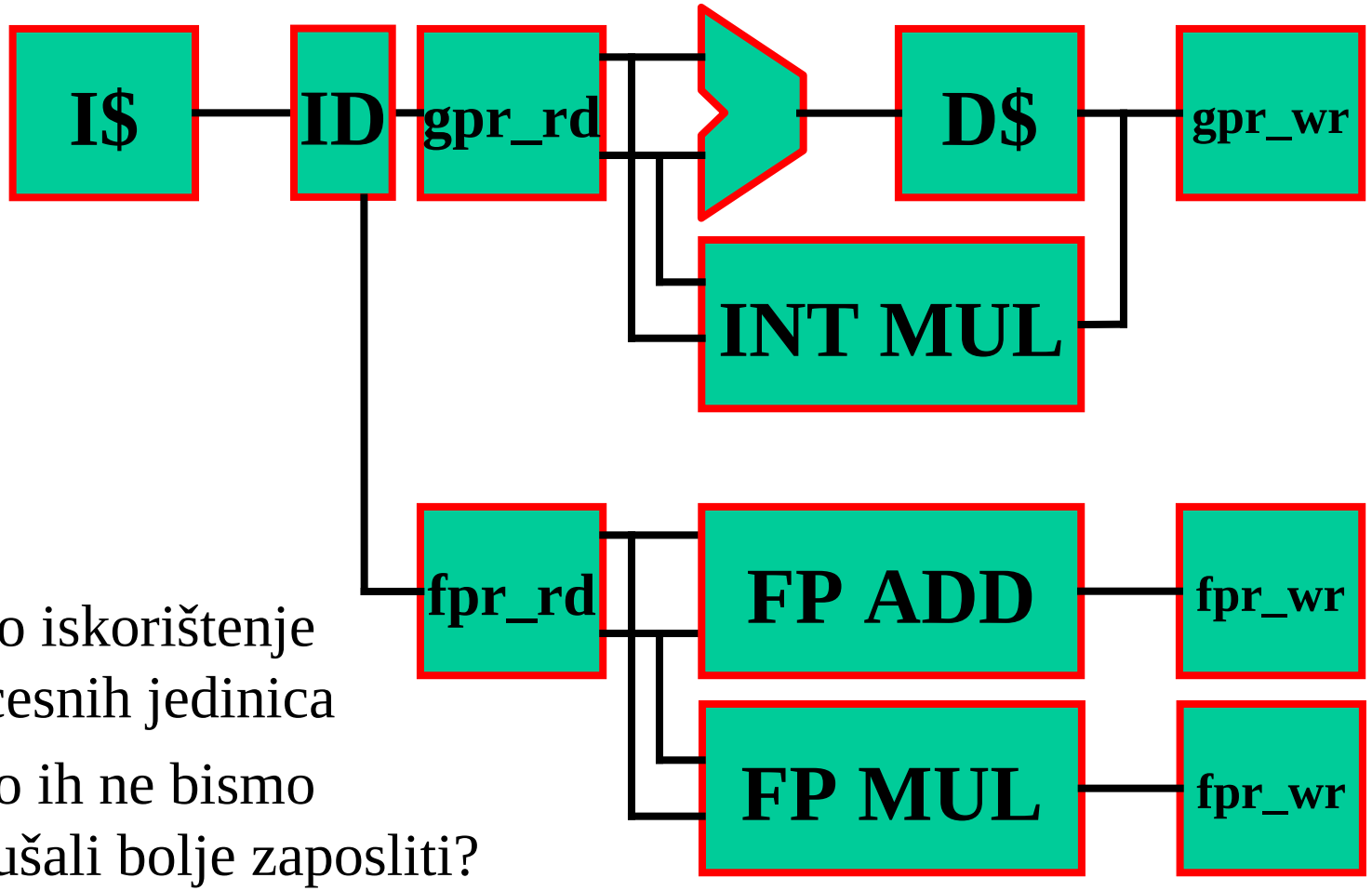
# Od protočnosti do višestrukog izdavanja




- 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 protočna organizacija s jednostrukim izdavanjem:**

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



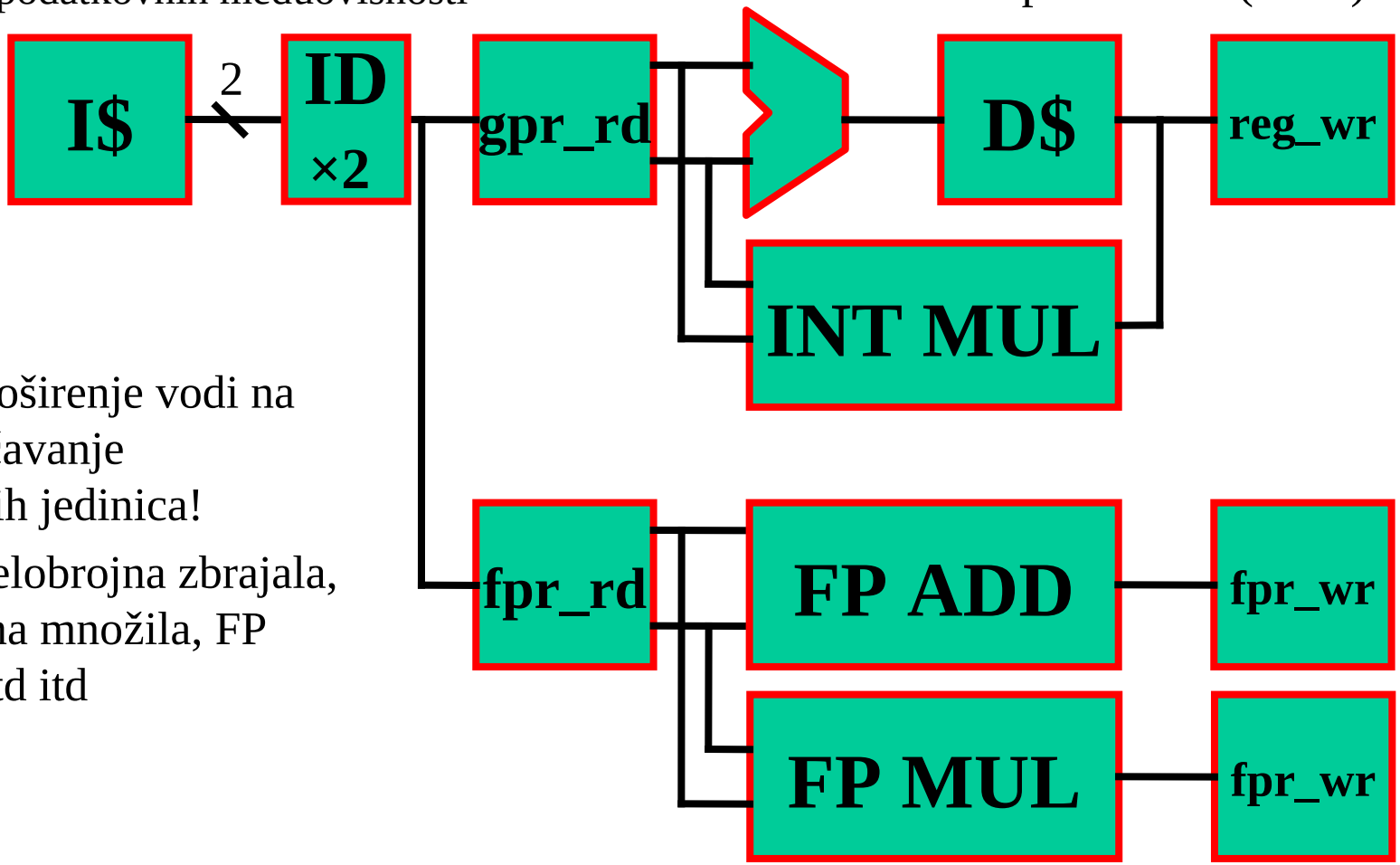

- slabo iskorištenje procesnih jedinica
- zašto ih ne bismo pokušali bolje zaposliti?

# 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

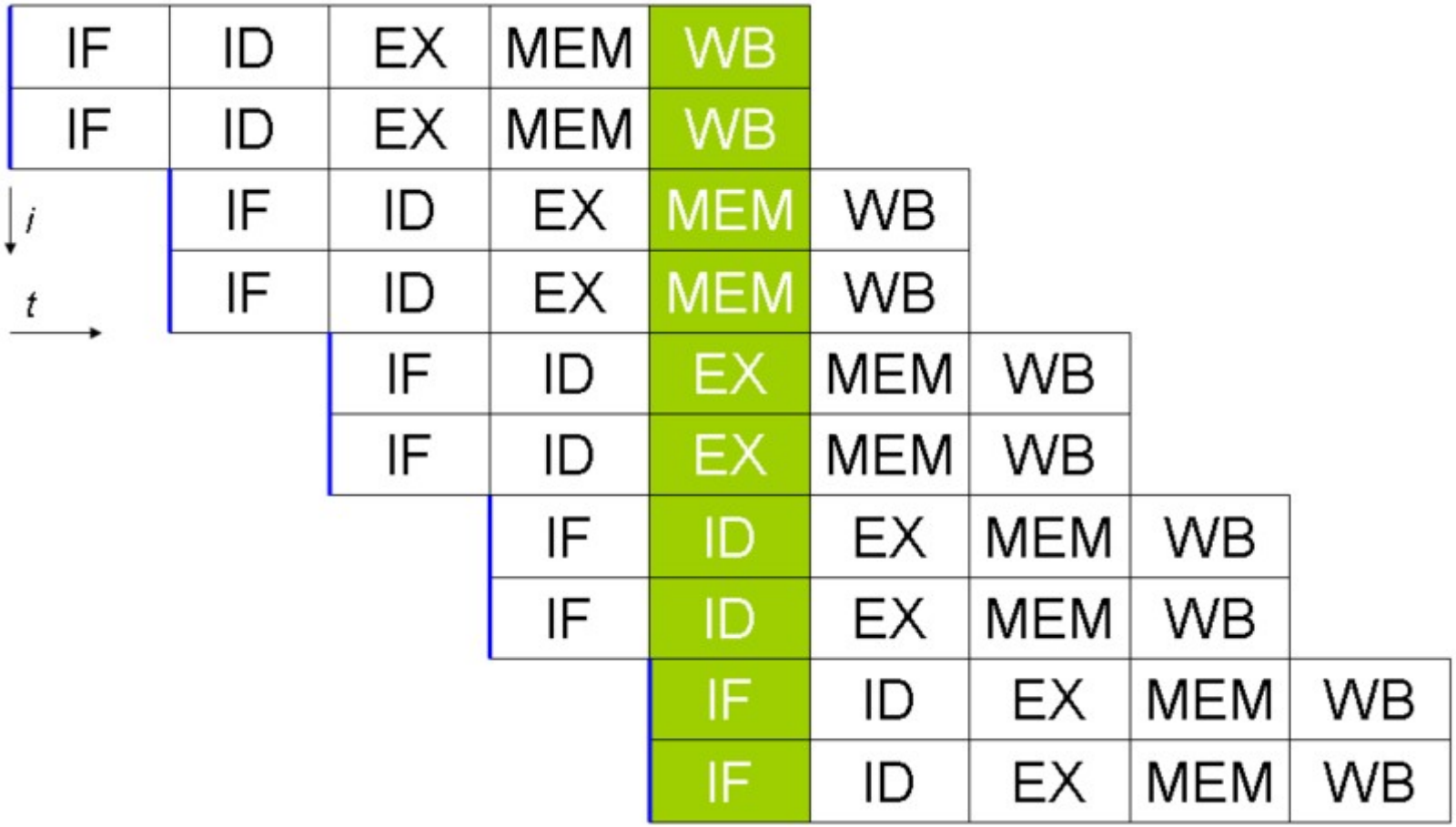
**Ideja**: istovremenim **izdavanjem** više instrukcija razotkriti više instrukcijskog paralelizma nego što to omogućava protočnost

- u implementaciji: replicirane protočne funkcijske 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:

- redosljed 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×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

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




[Wikipedia]

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

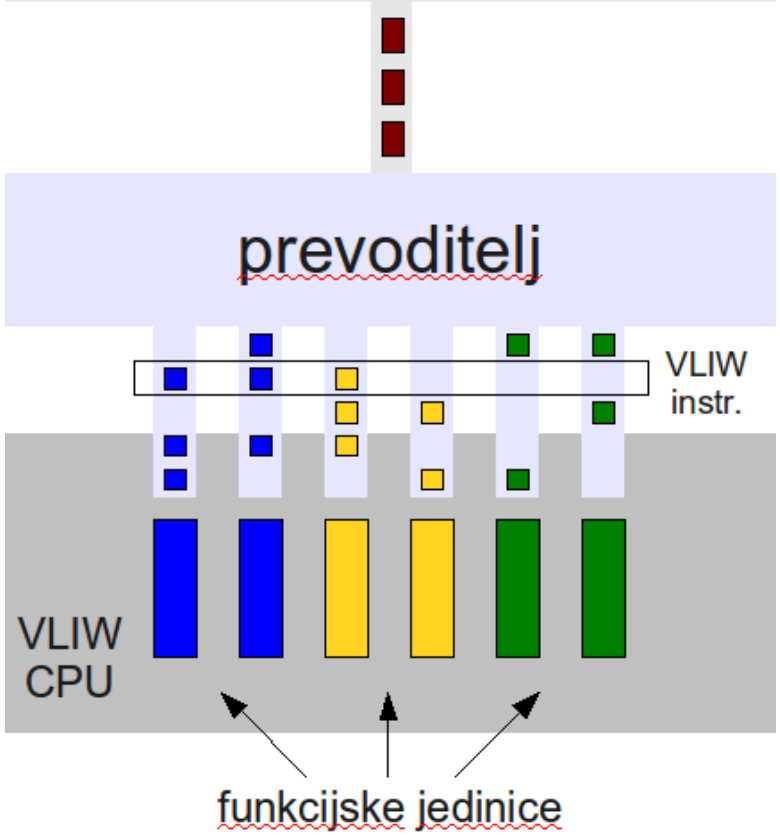
2. **dinamičko raspoređivanje** (“superskalarni” RISC)
   - procesor analizira instrukcijski tok i dinamički odabire instrukcije koje će se izdati u sljedećem ciklusu
   - procesor dinamički otkriva i razrješava hazarde (koncept upravljanja temeljenog na toku podataka)
   - prevoditelj pomaže prikladnim 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 međuovisnosti 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 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 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 ⇒ 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 sklopovlje 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 pretpostavimo 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 popunjenost protočnih jedinica.

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

Optimirani strojni kod (MIPS, 2× statičko 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

[Patterson08]

# 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/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 (vršni CPI = 0.5)

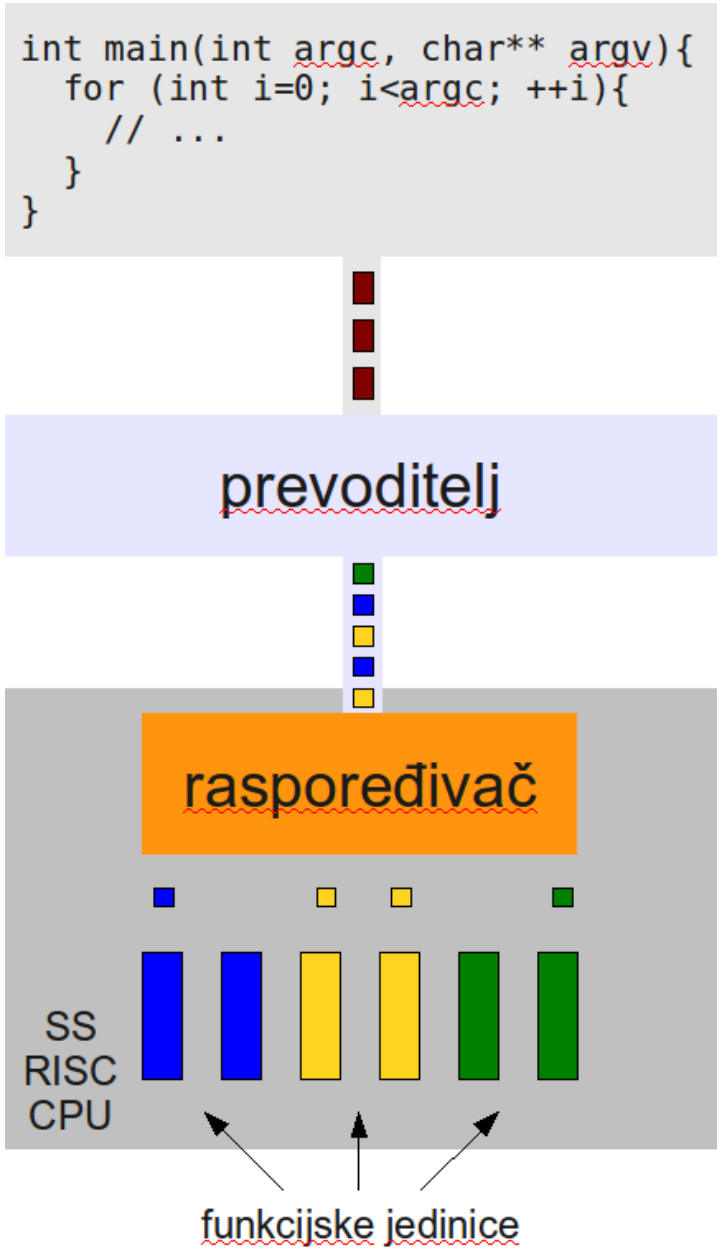
[Patterson08]

$n_I$ = 3.5/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



# 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 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
- ⇒ najbolje rezultate postiže dinamičko raspoređivanje!

# Motivacija za dinamičko raspoređivanje

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

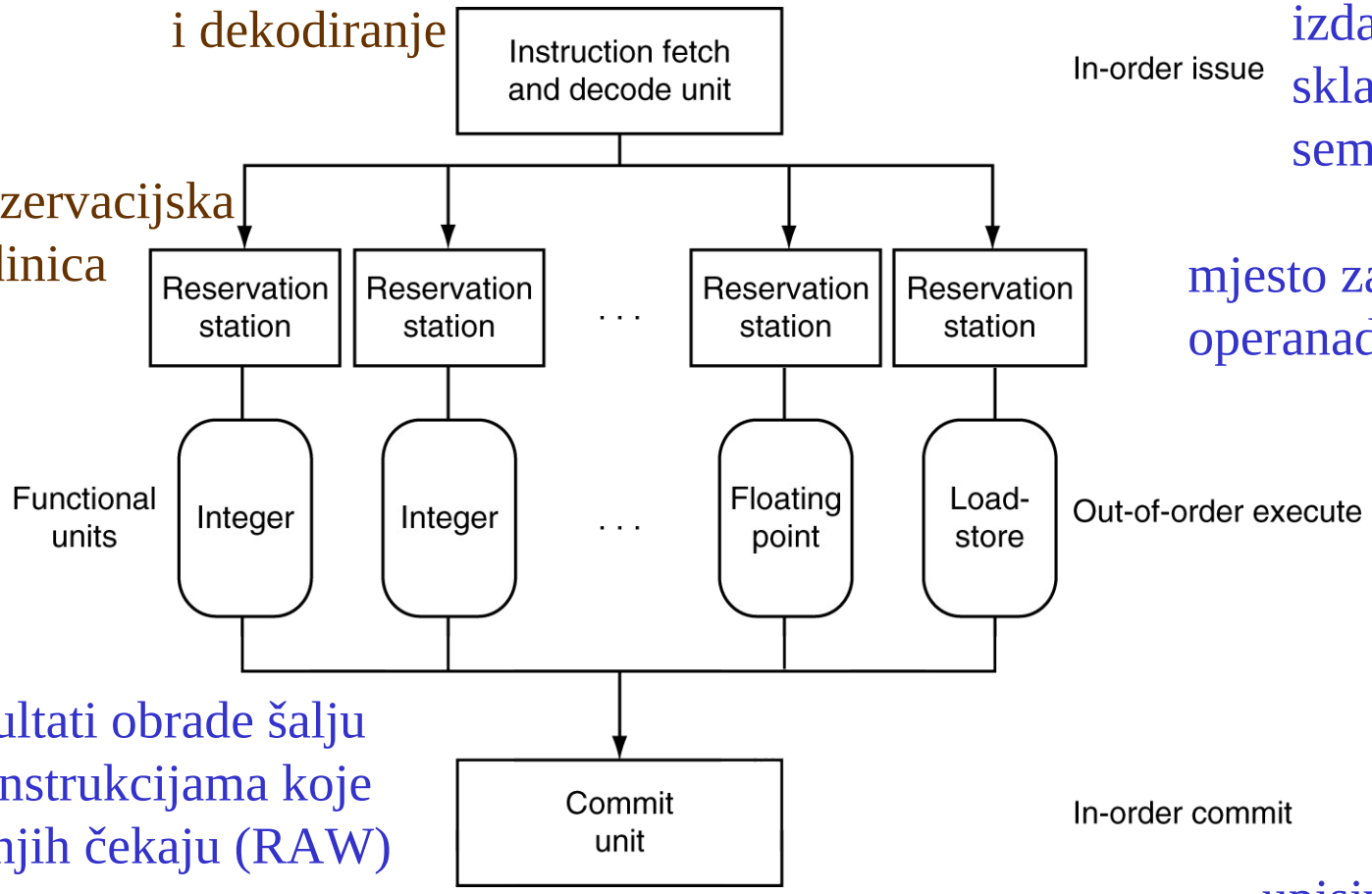
Međutim, sve hazarde nije moguće predvidjeti statičkom analizom:

- 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

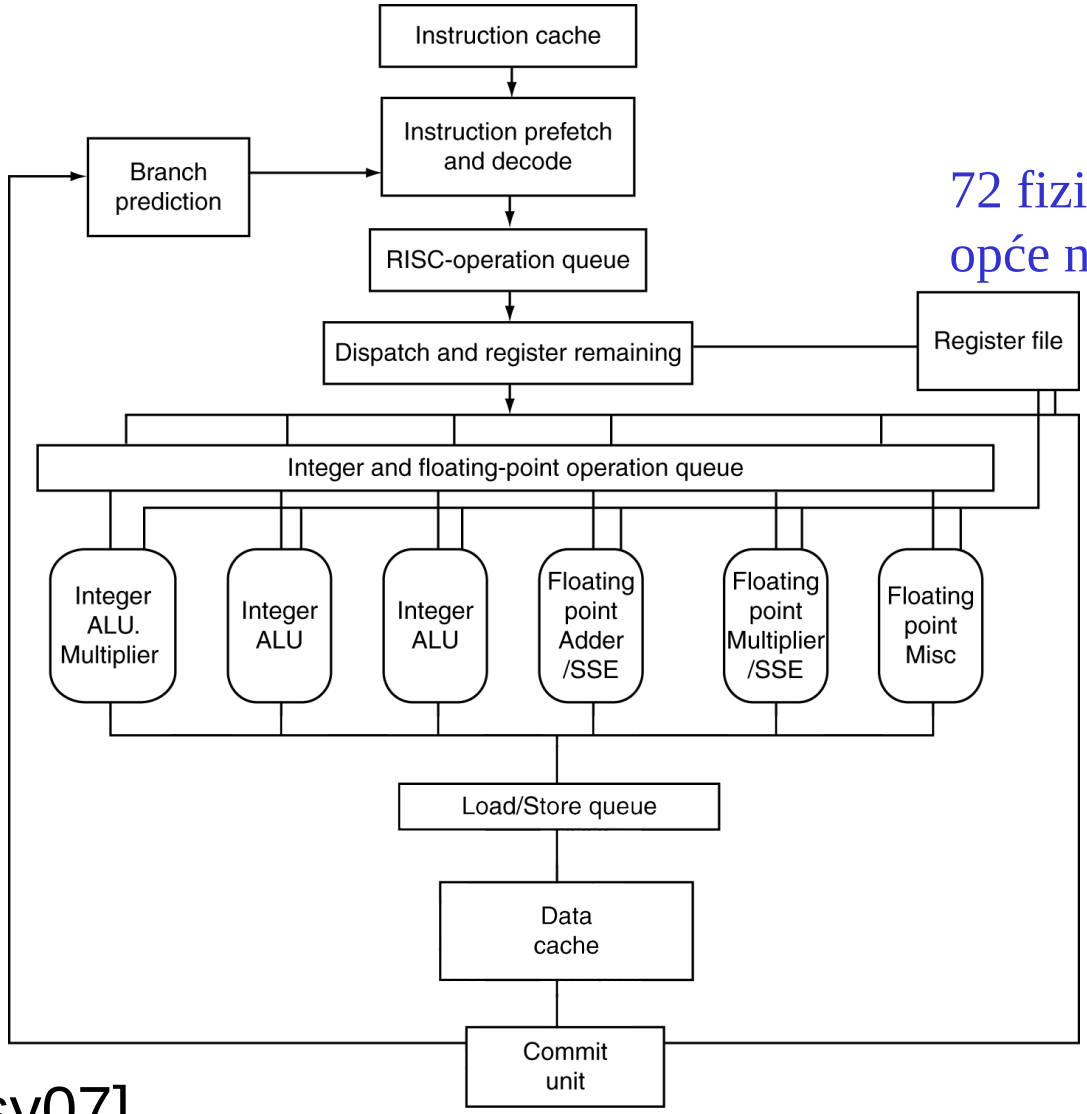
rezultati obrade šalju se instrukcijama koje na njih čekaju (RAW)

Upisna jedinica (reorder buffer)

upisivanje u odredišne registre u skladu s originalnim redosljedom

[Hennessy07]

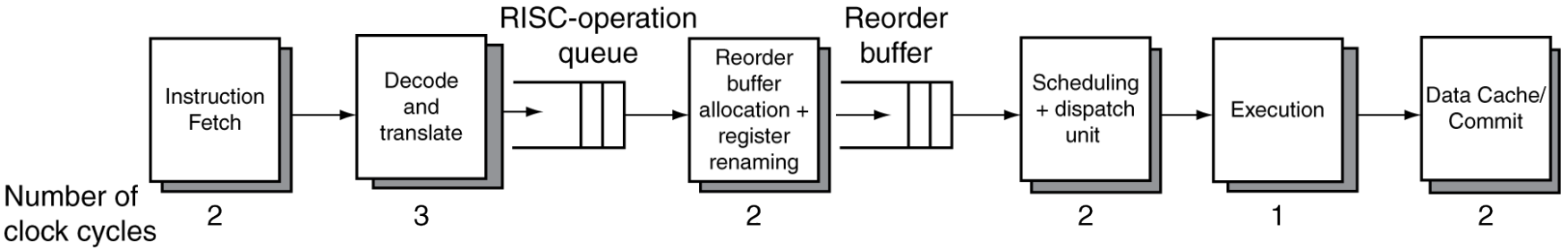
# Organizacija procesora Opteron X4




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?

**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];
}
```