

# Ostale teme za superskalare i chip multiprocesore

# Ubacivanje instrukcija u prozor

- In order, pa to radi programski brojač, ako se nije skočilo
- Skakanje – na osnovu predikcije!
- Širina reči instrukcijskog keša danas – 128 do 192 bita – tipično nekoliko instrukcija
- U slučaju da je RISC, tada su cele instrukcije u reči (bloku)
- Ako je CISC tada ne moraju instrukcije na početku i na kraju široke reči da budu cele!!!
- Moramo da znamo granice instrukcija, a to se može uraditi tek nakon bar delimičnog dekodovanja!!

# Šta se ubacuje po ciklusu u prozor - **RISC**

Pravila:

1. Ako je PC pokazao na početak bloka i nema skokova, učitavaju se dalje sve instrukcije iz bloka i šalju na paralelno dekodovanje
2. Ako je PC pokazao početak bloka, a u bloku je 1 instrukcija skoka => ako je predikcija da nema skoka učitavaju se sve instrukcije, a ako je predikcija da ima, učitavaju se sve instrukcije do instrukcije skoka i instrukcija skoka, a ostale ne ulaze u prozor
3. Ako je doskakanje negde u sredinu bloka, ne učitava se deo do tačke uskakanja, a za ostatak važi 2., ali od tačke uskakanja

# Dva skoka u bloku?

- Kompajler može da pravi razmak između instrukcija skoka (selidbe operacija) od najmanje onoliko instrukcija koliko ih ima u bloku
- Limit – zbog jedne predikcije po ciklusu, maksimalni instrukcijski paralelizam je 6-9 instrukcija **po ciklusu!** (Samo DoAll traže više)
- Eksperimenti sa prediktorima koji rade dve predikcije po ciklusu

# Šta raditi sa CISC?

- Lepiti delove instrukcija iz susednih blokova, jer zauzimaju 1-17 bajta!
- Raditi paralelno određivanje granica instrukcija u blokovima
- Kako to uraditi? Prediktor nam dozvoljava da se zalećemo i da radimo prefetching i delimično dekodovanje većeg broja reči instrukcijskog keša u kompleksni FIFO (in order issue) zbog paralelnog određivanja **granica instrukcija**
- Radimo prefetching većeg broja bazičnih blokova na dinamičkom tragu još pre instrukcijskog prozora!!!

# Posledice dohvatanja većeg broja bazičnih blokova

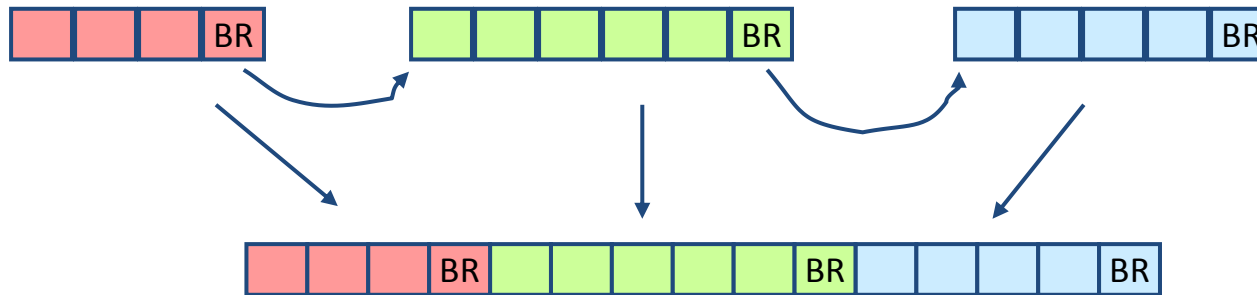
- Potreban je trace multiportni cache za pamćenje dinamičkog traga unapred
- Multiportni, jer se paralelno čita na više dekodera da bi se dobile mikrooperacije
- to dodaje protočne stepene i povećava kaznu zbog greške u predikciji – nije previše značajno
- Imamo ograničenje u propusnosti zbog grananja

# Predobrada za instrukcijski prozor

- Rešenje: dohvatati reči sa instrukcijama unapred nekoliko bazičnih blokova i obezbediti brže određivanje granica instrukcija nego što će se na kraju ubacivati u prozor – Intel 50% više
- Sve se to radi u pipeline-u: prediktor određuje šta se ubacuje, u pipeline-u se određuju granice instrukcija i zatim radi dekodovanje, a na kraju postoji i queue mikroinstrukcija koje se tek onda ubacuju u ROB, odnosno instrukcijski prozor (ugrađenu dataflow mašinu)

# Trace Cache

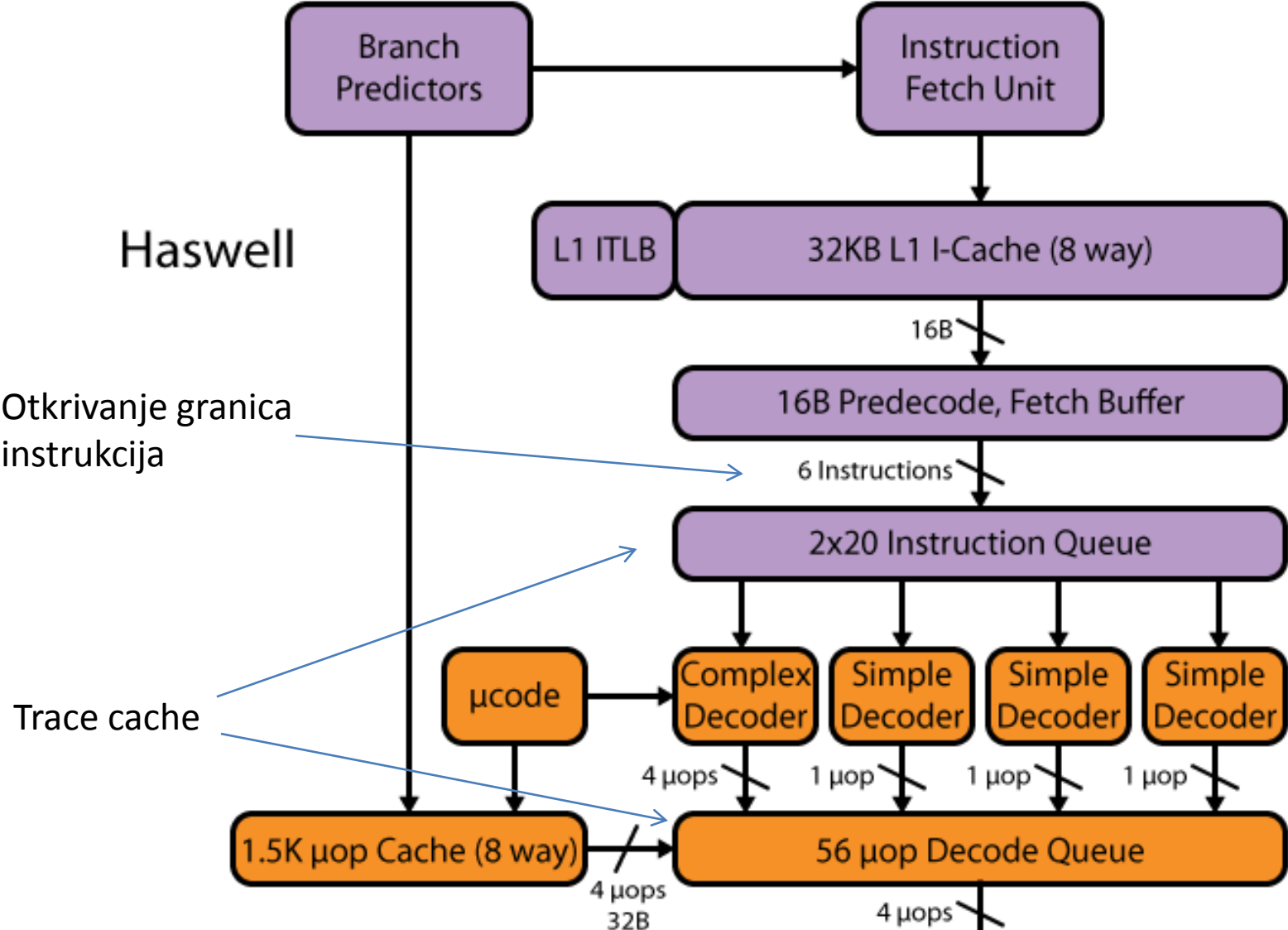
- Ključna ideja: Pakovati više nesusednih bazičnih blokova u jednu susednu keš reč



- Jedno dohvaćanje je dohvaćanje više bazičnih blokova – gube se granice bazičnih blokova
- Trace cache indeksira na osnovu startne adrese i sledećih n predikcija grananja
- Korišćeno od Pentium 4 procesora za čuvanje dekodovanih mikrooperacija



# Trace Cache za Haswell



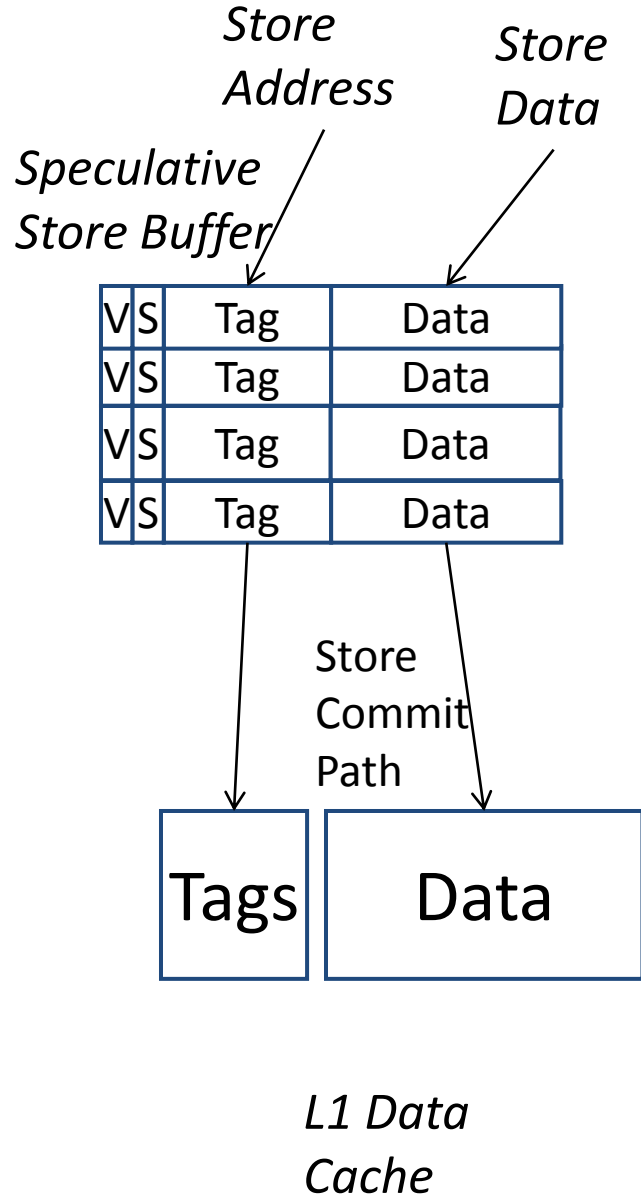
# Zavisnosti po podacima preko memorije

- Prave zavisnosti, antizavisnosti i izlazne zavisnosti, jer nema dinamičkog preimenovanja za memoriju
- Store instrukcije treba po originalnom redosledu izvršavati i tek kada su in order (commit) – mali ROB za store, pa zbog redosleda nema izlaznih zavisnosti
- Antizavisnosti praktično ne mogu nastati, ako postoji load buffer (load nije zavisan po podacima iz registara, pa kreće odmah u izvršavanje ako nema pravih zavisnosti preko memorije)
- Ali tada imamo store – load – store problem

# Load-Store redovi

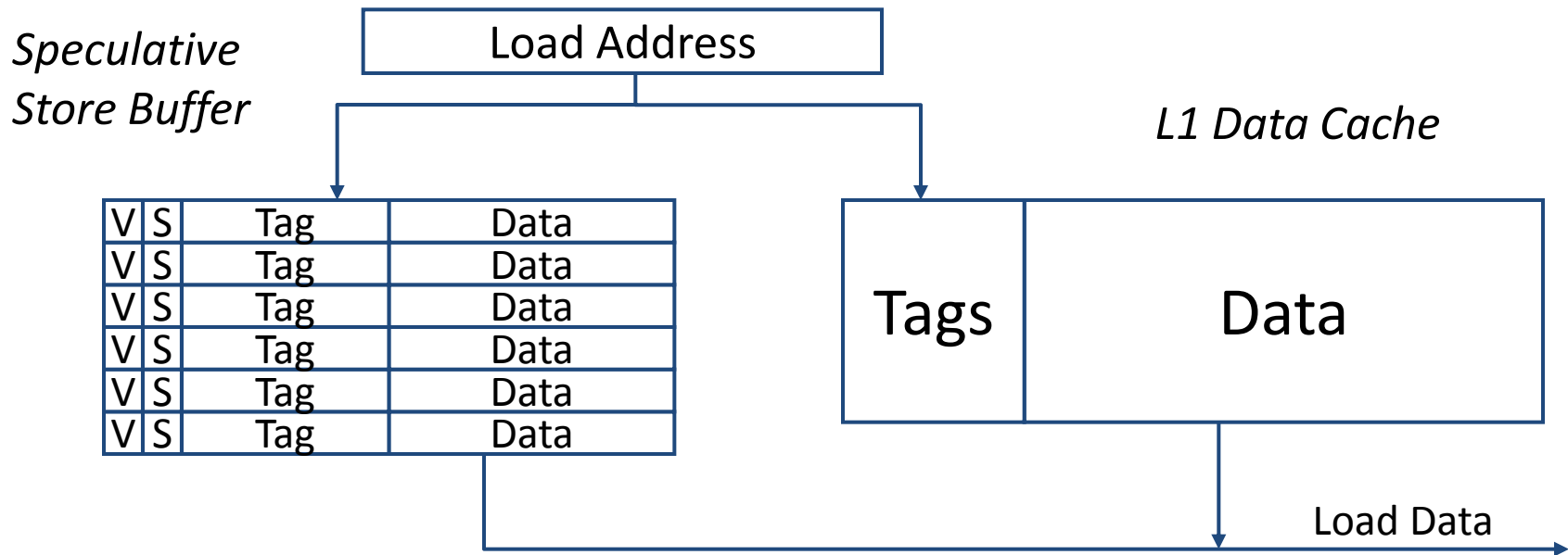
- Zavisnosti preko memorije bi mogli da značajno ograniče performanse, zbog velike dužine pipelina za Load i Store, upisa tek kada je commit Store instrukcije i sporosti memorije
- Mora se ugraditi kompleksna load-store reordering tehnika da se smanji efektivno kašnjenje memorije dozvoljavanjem spekulativnih load-a
- Zašto su spekulativni? – ne znamo da li neka ranija store instrukcija ne menja sadržaj lokacije, jer može još da se izračunava adresa

# Spekulativni Store Buffer



- Ne sme se menjati sadržaj memorije dok store instrukcija nije komitovana. Zato spekulativni store buffer čuva podatke spekulativnih store podataka
- Tokom dekodovanja se redom zauzimaju ulazi (lokacije) po programskom redosledu
- Store operacija se deli na "store address" i "store data" mikro-operacije - "Store address" izvršavanje upisuje tag, a "Store data" izvršavanje upisuje podatke
- Store se komituje kada se podaci i adresa za najstariji store upišu i instrukcija je komitovana (ROB):
  - Ukloni se 1 za spekulativni bit S i komitovani podaci se upisuju u cache
- Moguć store abort reset valid bita V

# Load bypass iz spekulativnog store buffera



- Ako i store buffer i cache imaju podatak, koji uzeti?  
**Spekulativni store buffer**
- Ako je dva puta ista adresa u store bufferu, koji upis uraditi?  
**Najnoviji store (jer je podatak ranije store operacije pokupljen iz spekulativnog store buffer-a)**

# Memorijske zavisnosti

`sd x1, (x2)`

`ld x3, (x4)`

- Kada se može raditi load?

# In-Order Memory Queue

- Sve Load i Store operacije moraju po programskom redosledu
- => Load i Store ne mogu da napuste ROB dok sve prethodne Load i Store operacije ne završe izvršavanje
- Load i Store mogu spekulativno u odnosu na ostale instrukcije

# Konzervativno O-o-O Load Izvršavanje

```
sd x1, (x2)
ld x3, (x4)
```

- Može load pre store-a ako se za adrese zna da je **x4 != x2**
- Svaka load adresa se poredi sa adresama svih prethodno nekomitovanih store operacija
- Ne raditi load, ako je bilo koja od prethodnih adresa store-a nepoznata

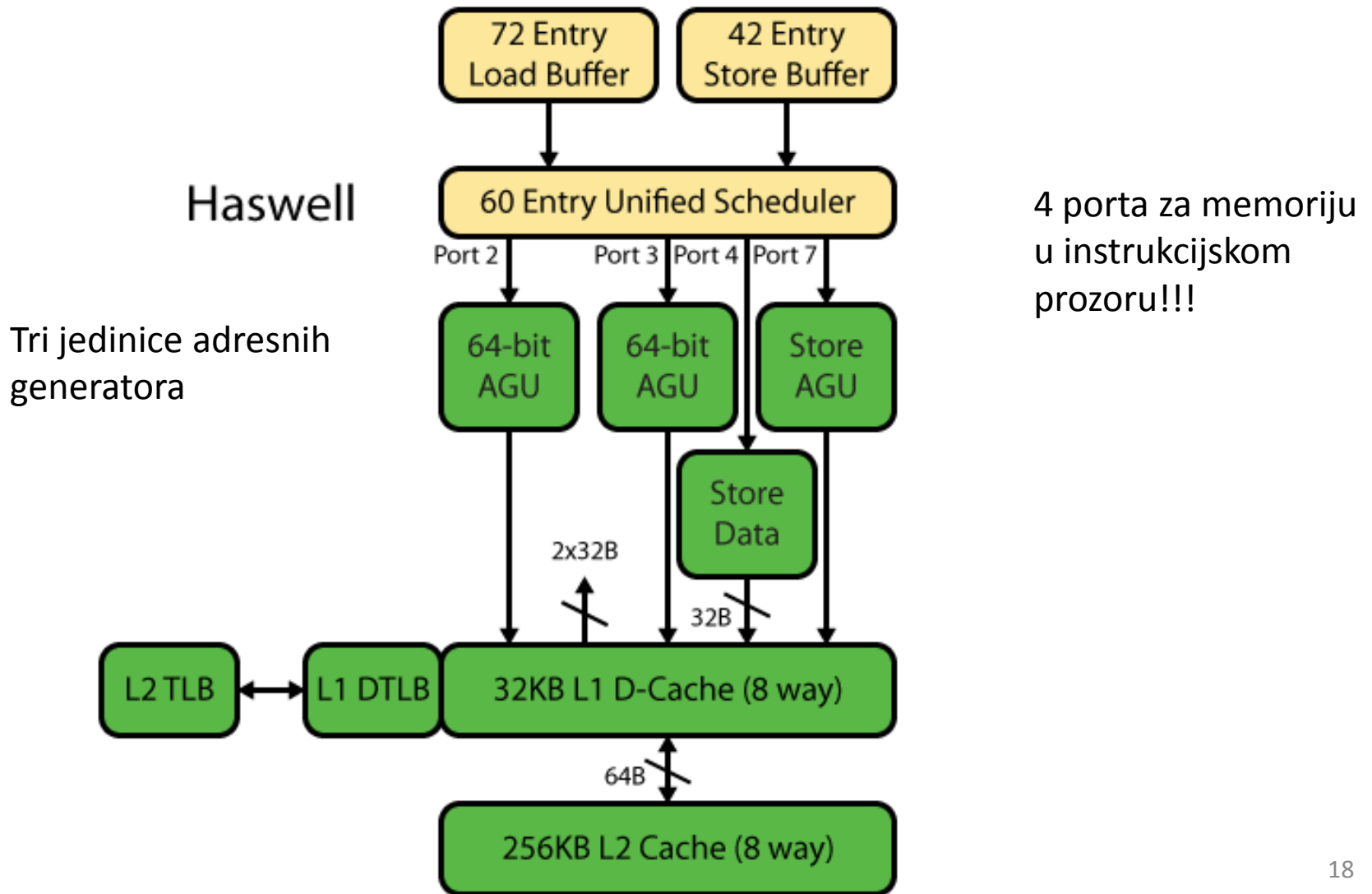


# Adresna Spekulacija

```
sd x1, (x2)  
ld x3, (x4)
```

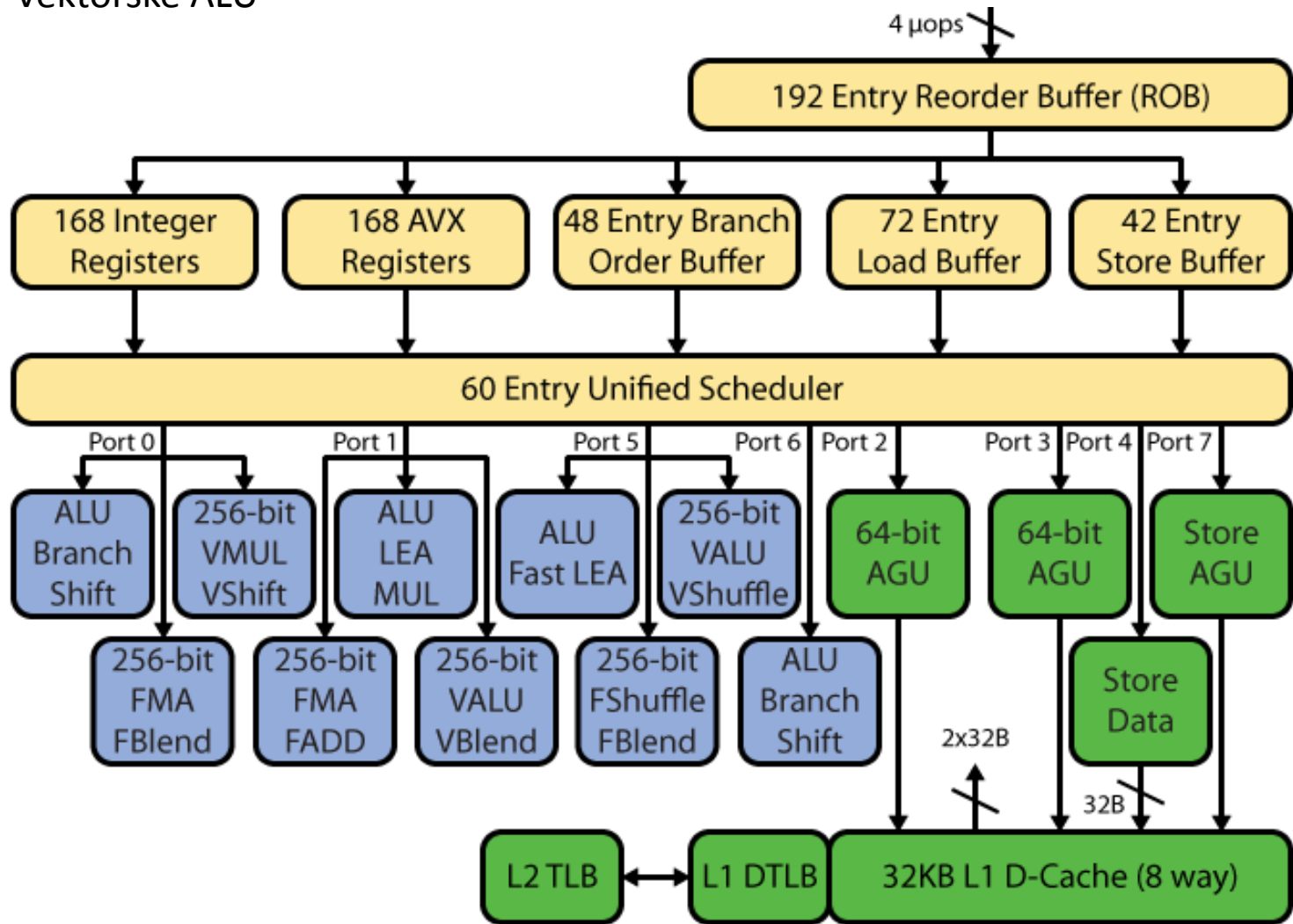
- Pretpostavimo da je  $x4 \neq x2$
- Izvrši se load pre nego što je poznata store adresa
- Moraju se čuvati sve nekomitovane load/store adrese u programskom redosledu
- Ako se dogodi da je  $x4 == x2$ , obustavi load i sve prateće instrukcije (load exception)
- => Velika kazna za grešku zbog netačne pretpostavke o različitosti adresa

# Load – Store kod Haswell



# Haswell prozor, registri i FJ

Vektorski registri AVX i  
vektorske ALU



# Multithreading i Multicore

# Šta razmatramo kod Multithreading-a

- Pregled Threading Algoritama
- Hyper-Threading Koncepti
- Hyper-Threading Arhitektura
- Prednosti i mane

# Threading Algoritmi

- Time-slicing
  - Procesor se prebacuje između niti u fiksnim vremenskim intervalima.
  - Visoka cena, naročito ako je jedan od procesa u stanju čekanja. **Fine grain**
- Switch-on-event
  - Niti se menjaju u slučaju da se događa bilo koje čekanje u niti koji se izvršava
  - Ako se čekaju podaci sa sporog izvora, CPU se predaje nekom od drugih procesa (ne OS). **Coarse grain**

# Threading Algoritmi (2)

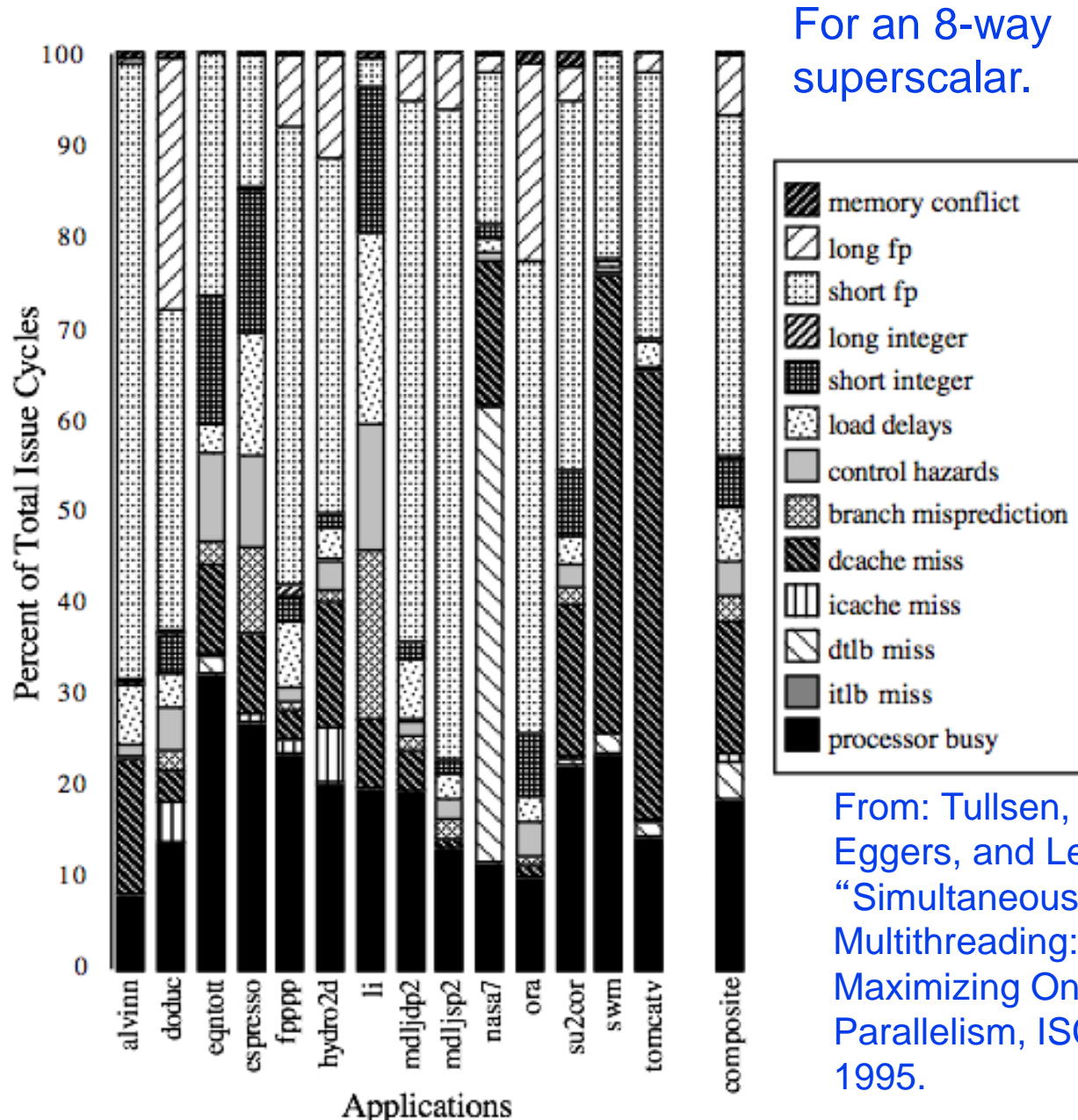
- Multiprocesiranje
  - Ukupan posao se distribuira na više procesora
  - Značajan dodatni trošak
- Simultaneous Multi-Threading (SMT)
  - Više niti se izvršavaju na jednom procesoru bez izmene konteksta
  - Osnova za Intelovu Hyper-Threading tehnologiju.

# Hyper-Threading Concept

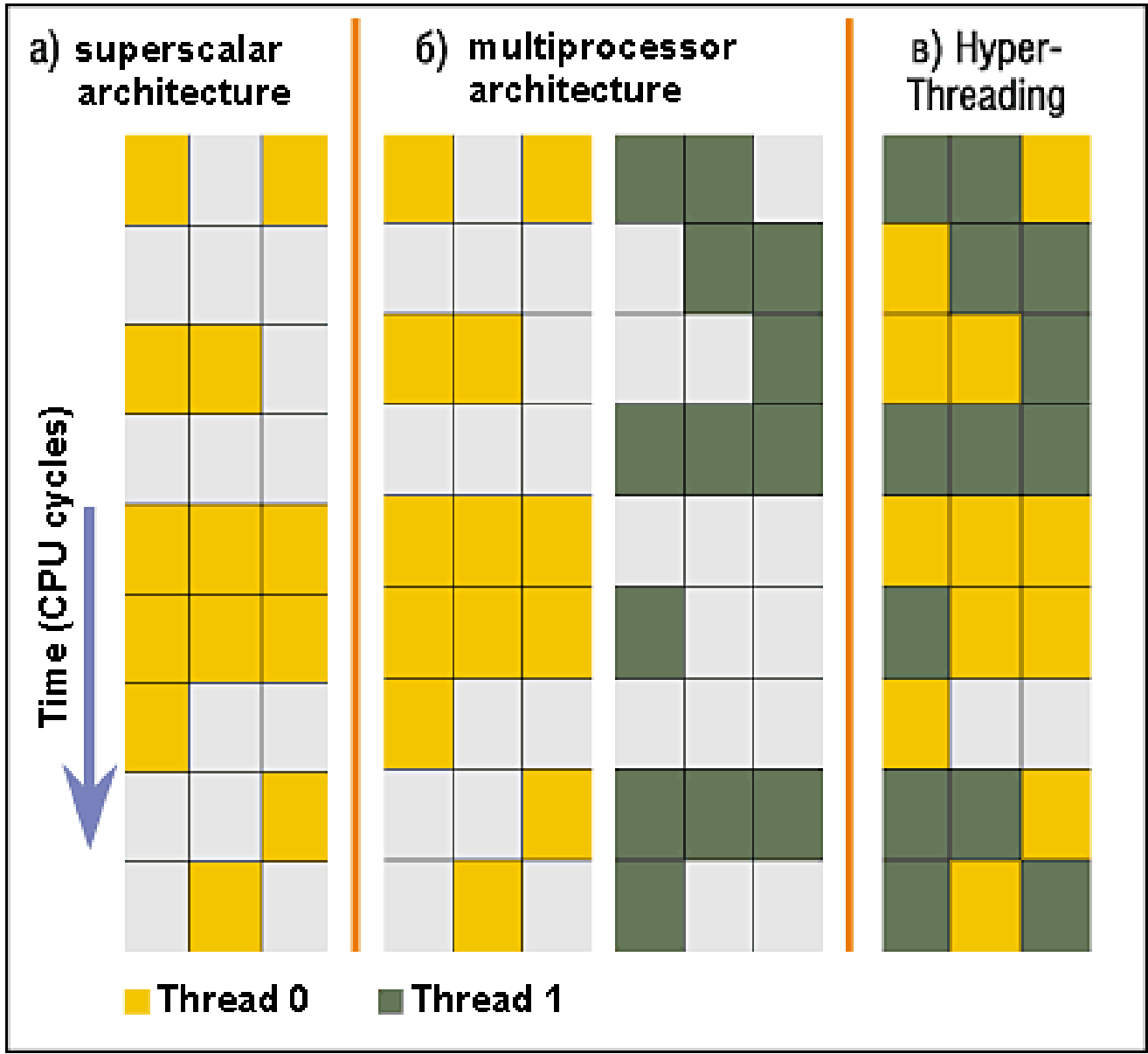
- U svakom trenutku u vremenu, samo deo procesorskih resursa se koristi za izvršavanje programskog kôda niti
- Neiskorišćeni resursi mogu biti istovremeno iskorišćeni za paralelno izvršavanje druge niti/aplikacije
- Kod servera ili klijentskih računara često postoji puno niti, pa je veoma korisno

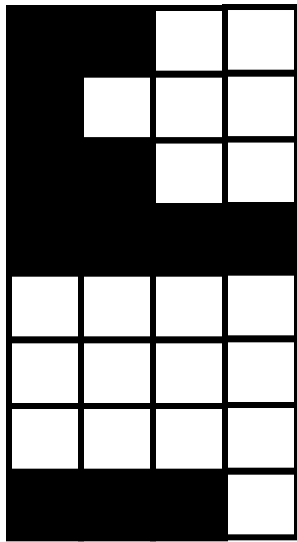


# Quick Recall: Many Resources IDLE!

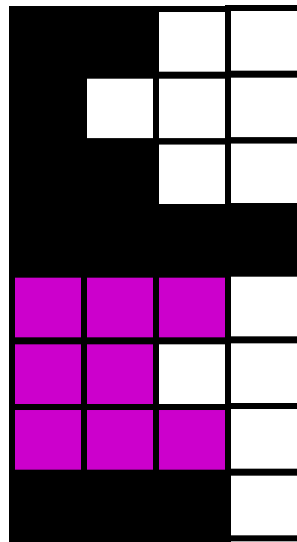


From: Tullsen, Eggers, and Levy, "Simultaneous Multithreading: Maximizing On-chip Parallelism, ISCA 1995.

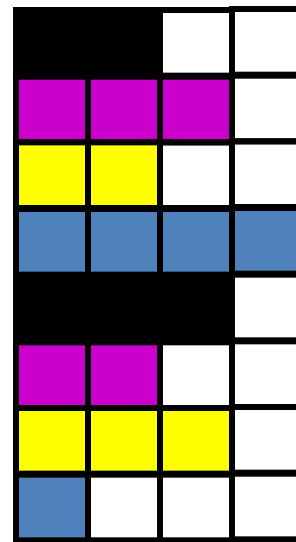




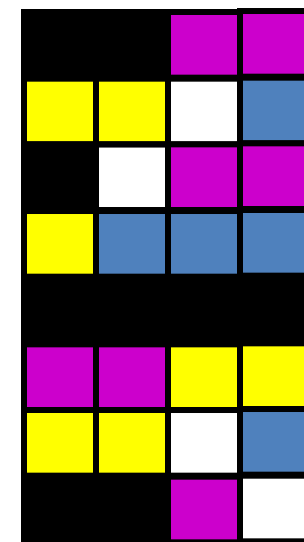
(a)



(b)



(c)



(d)

(a) Superskalarni procesor bez multithreading-a

(b) Superskalarni procesor sa coarse-grain multithreading-om

(c) Superskalarni procesor sa fine-grain multithreading-om

(d) Superskalarni procesor sa simultaneous multithreading-om (SMT)

# Simultaneous Multithreading (SMT)

Primer: novi Intel procesori sa “Hyperthreading” - om

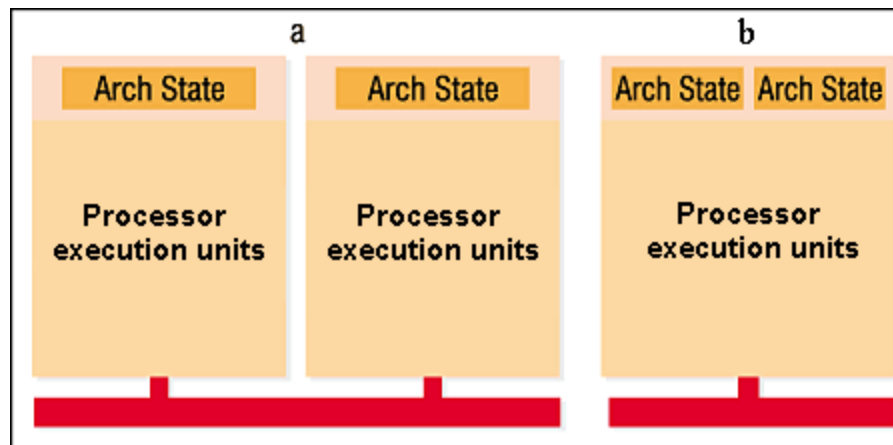
Osnovna ideja: Iskoristiti instrukcijski nivo paralelizma nad više niti istovremeno; tj. pretvoriti paralelizam na nivou niti u dodatni instrukcijski nivo paralelizma

Iskoristiti sledeće osobine modernih procesora:

- Više funkcionalnih jedinica, pa postoji znatan višak FJ u odnosu na prosečne potrebe jedne niti
- Preimenovanje registara i dinamičko raspoređivanje (ugrađena data flow mašina) - Više instrukcija iz više nezavisnih niti mogu da koegzistiraju i da se istovremeno izvršavaju, uz veći ukupni paralelizam dva smanjena nepovezana dinamička DDG!

# Hyper-Threading Arhitektura

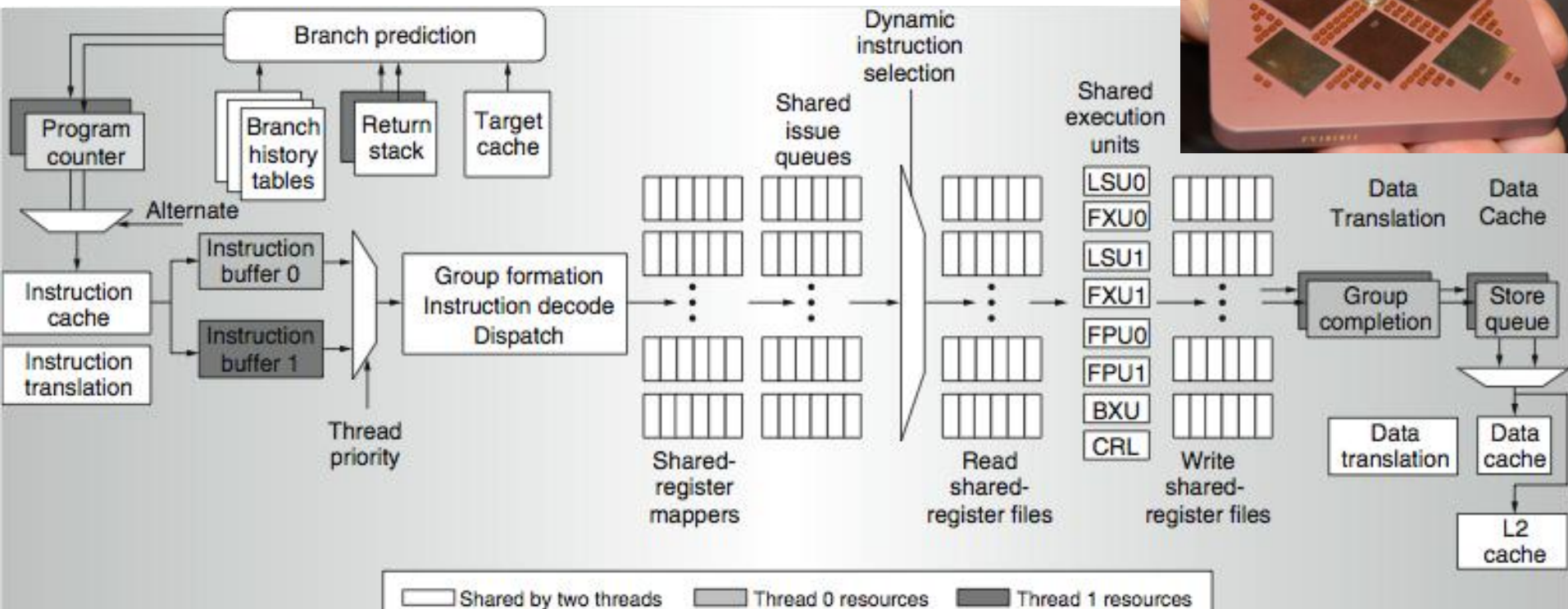
- Prvo se pojavila kod Intel Xeon MP procesora
- Stvara iluziju kojom se jedan fizički procesor javlja kao više (2) logička procesora
- Svaki logički procesor ima kopiju arhitekturnog stanja
- Logički procesore dele jedan zajednički skup fizičkih resursa za izvršavanje



# Hyper-Threading Arhitekture

- Operativni sistemi i korisnički konkurentni programi mogu da rasporede procese ili niti na logičke procesore kao da se raspoređuju na multiprocesorski sistem sa istim brojem fizičkih procesora
- Kako logički procesori mogu da koriste deljene resurse:
  - Cache, izvršne jedinice, Prediktore grananja, kontrolnu logiku, OoO logiku, magistrale, fizičke registre, ...

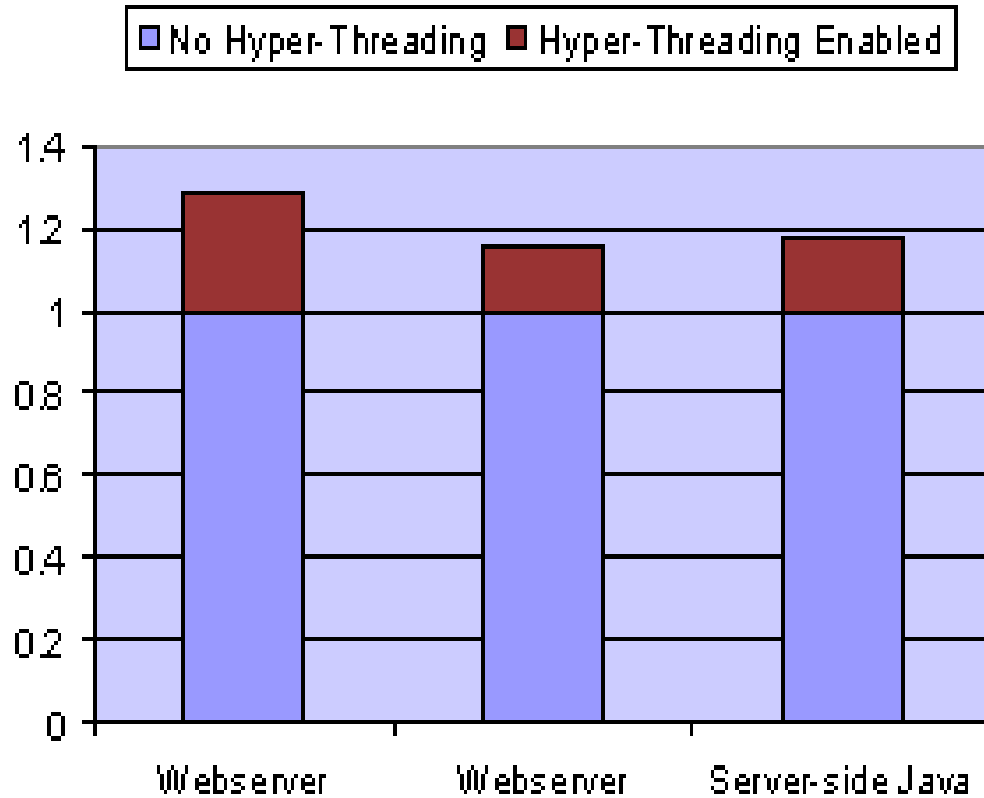
# Power 5 dataflow ...



- **Zašto samo dve niti?**
  - Sa 4, neki od deljenih resursa (fizički registri, cache, memorijski propusni opseg) bi postao usko grlo
- **Cena:**
  - Power5 jezgro je oko 24% veći od Power4 jezgra zbog podrške za simultani multithreading

# Prednosti

- Kod nekih procesora je dodatna površina na čipu oko 5%
- Nema gubitka performansi ako je samo jedna nit aktivna, a poboljšane performanse sa dve niti (dva manja nepovezana DDG niti istovremeno)
- Bolje iskorišćenje resursa





# Mane

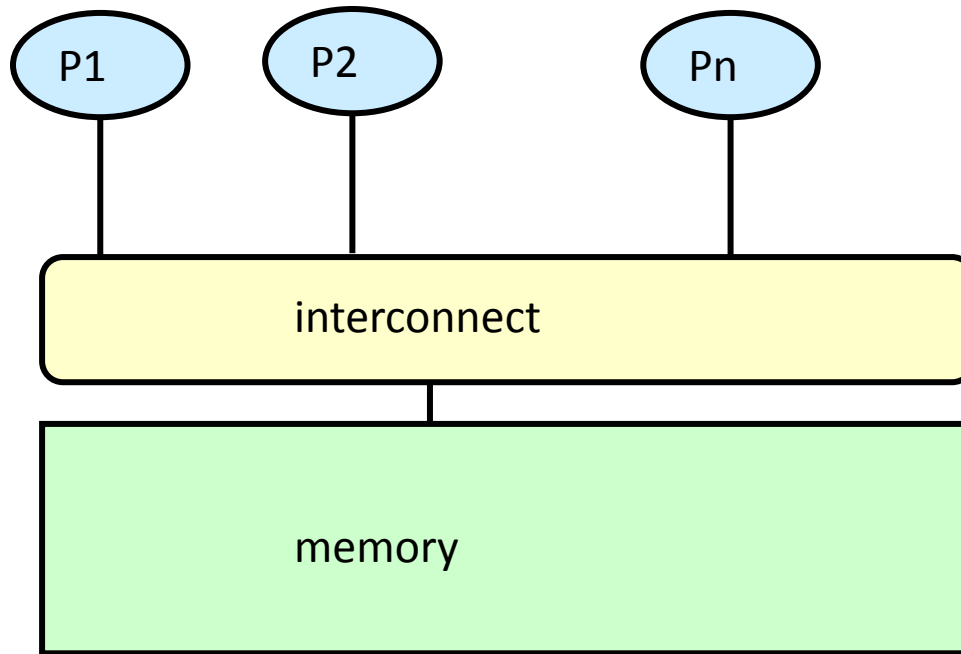
- Da se iskoristi, mora da bude konkurentni program
  - Niti nisu determinističke i mora pažljivi dizajn aplikacije
  - Niti imaju dodatnu kontrolnu logiku
- Konflikti deljenih resursa – cache i memorija pre svega
- Ukupni dobitak je mali, ali će rasti sa veličinama instrukcijskog prozora!

# Multicore

- Multiprocesori na čipu
- UMA multiprocesori (symmetric multiprocessing)

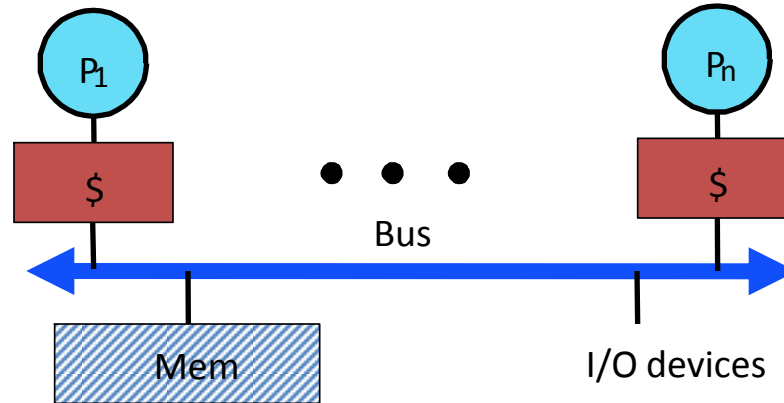
# Osnovna Shared Memory Arhitektura

- Procesori svi povezani na veliku deljenu memoriju
  - Gde su cache memorije?



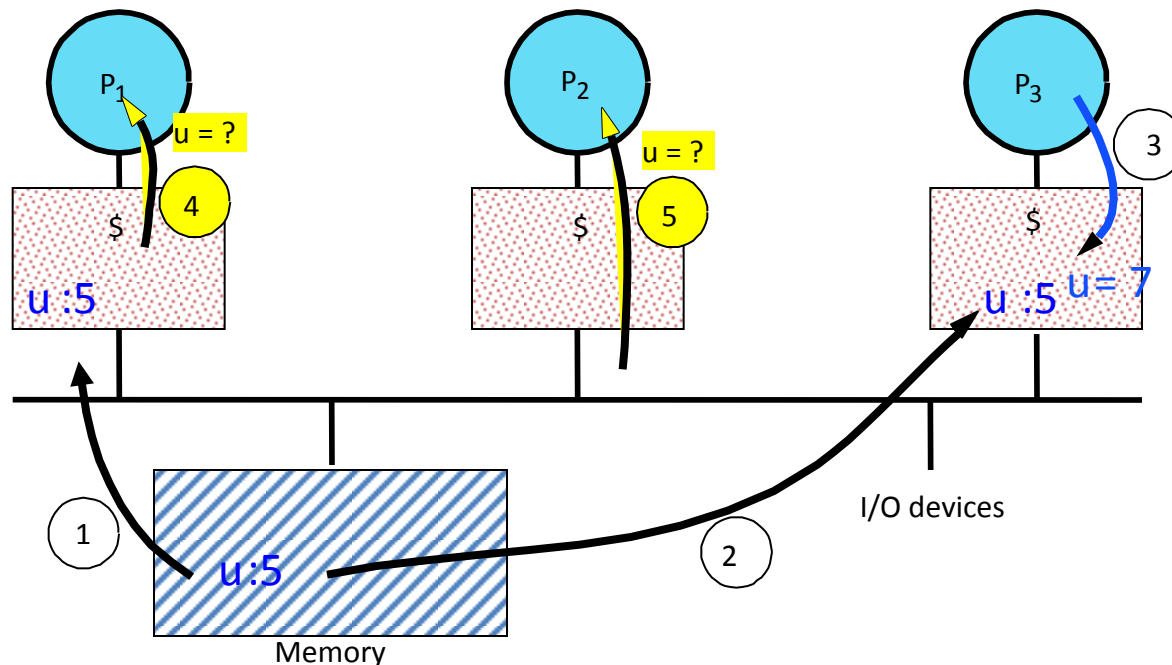
- Koja su ograničenja, kako su napravljeni, limiti, programiranje?

# Kako i gde Cache???



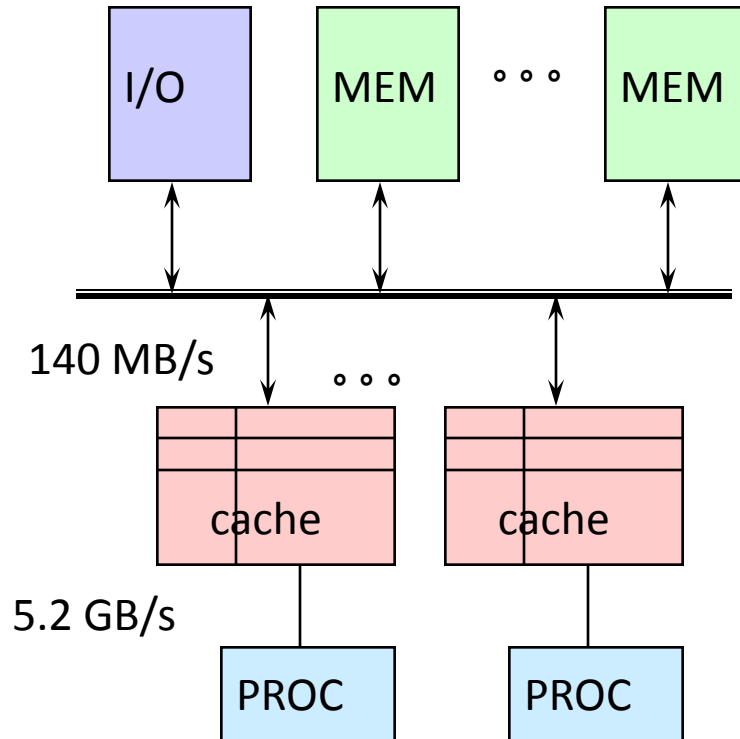
- Za visoke performanse sa deljenom memorijom: koristi cache-eve!
  - Svaki procesor ima jedan ili više svojih cache-eva
  - Stavi podatke iz memorije u cache
  - Writeback cache: nemoj slati sve podatke preko bus-a u memoriju
- Cache-evi smanjuju srednje kašnjenje memorijskog sistema
  - Automatska replikacija bliže procesoru
  - *Važnije* za multiprocesor nego za jedan procesor: veća su kašnjenja
- Normalan uniprocessorski mehanizam za dohvatanje podataka
- **Problem: Koherencija Cache-eva!**

# Primer problema koherencije Cache-a



- Processori mogu da vide različite vrednosti u nakon događaja 3
- Sa write back cache-evima, vrednost upisana u memoriju zavisi od toga koji cache i kada da upiše u memoriju
- Kako ovo popraviti na bus-u: Coherence Protocol
  - Upotrebiti bus da emituje (broadcast) upise ili invalidacije
  - Jednostavni protokoli zasnovani na broadcast-u na bus-u
  - Bus može do 32- 64 procesora (max)

# Ograničenja Bus-Based Shared Memory



Assume:

1 GHz processor w/o cache

=> 4 GB/s inst BW per processor (32-bit)

=> 1.2 GB/s data BW at 30% load-store

Suppose 98% inst hit rate and 95% data hit rate

=> 80 MB/s inst BW per processor

=> 60 MB/s data BW per processor

⇒ 140 MB/s combined BW

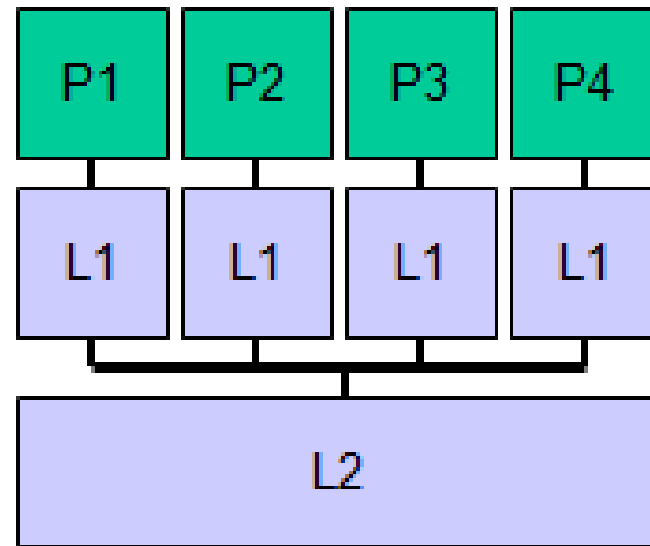
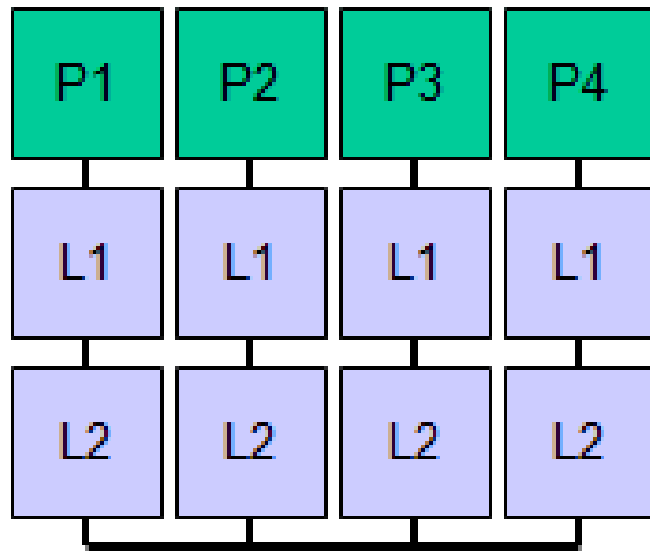
Assuming 1 GB/s bus bandwidth

∴ 8 processors will saturate bus

# Cache Organizations for Multi-cores

---

- L1 caches are always private to a core
- L2 caches can be private or shared – which is better?



# Cache Organizacija za Multi-core

---

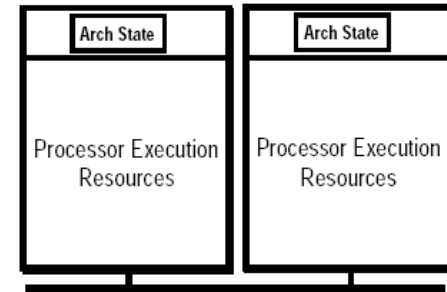
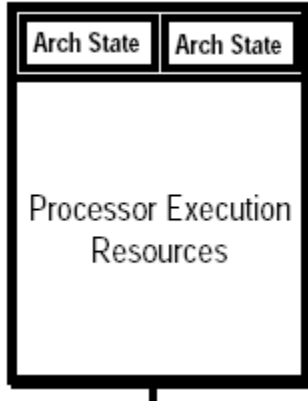
- L1 cache je uvek privatn za core
- L2 cache može biti privatn ili deljeni
- Prednosti deljenog L2 cache-a:
  - efikasna dinamička alokacija prostora za svaki core
  - Podaci deljeni od strane više core-ova se ne repliciraju
- Prednosti privatnog L2 cache-a:
  - brzi pristup privatnom L2 – dobro za male programe
  - privatn je bus do privatnog L2 → manje sačekivanja



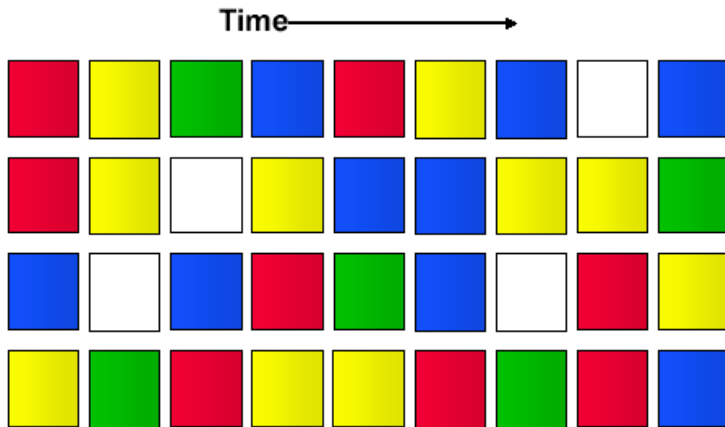
# Podsetnik: SMT

(Simultaneous Multi Threading)

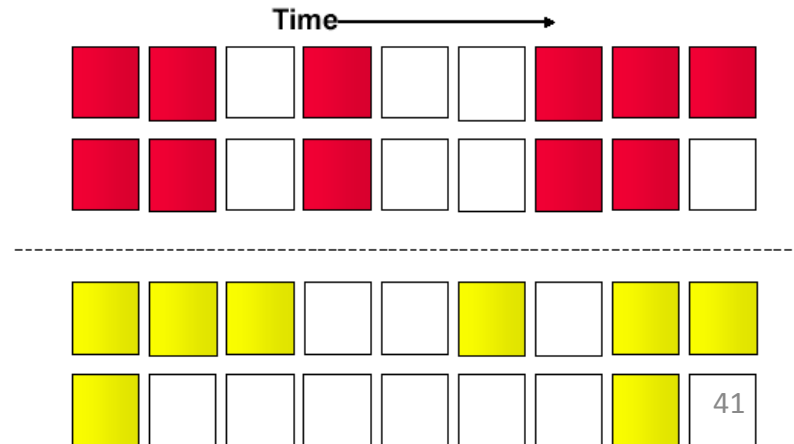
SMT ↘



**Simultaneous Multithreading**

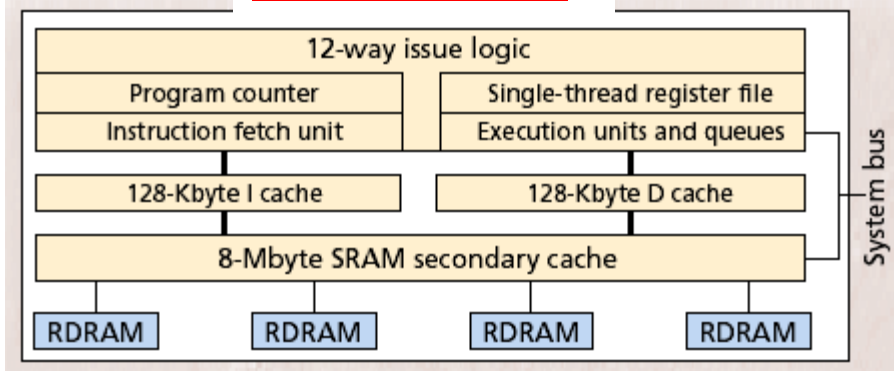


**CMP – Chip Multi-Processor**

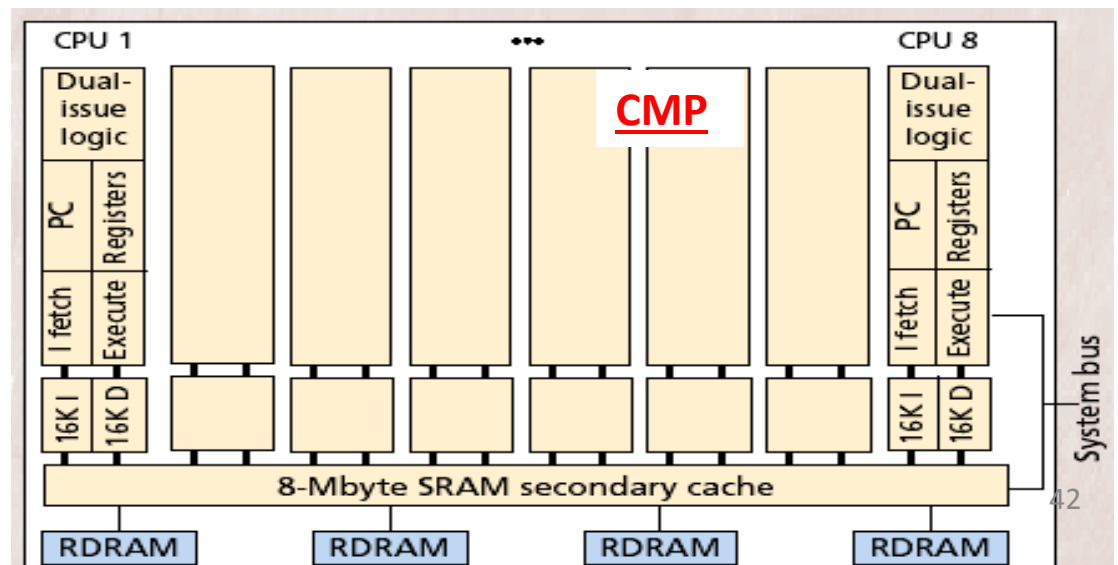
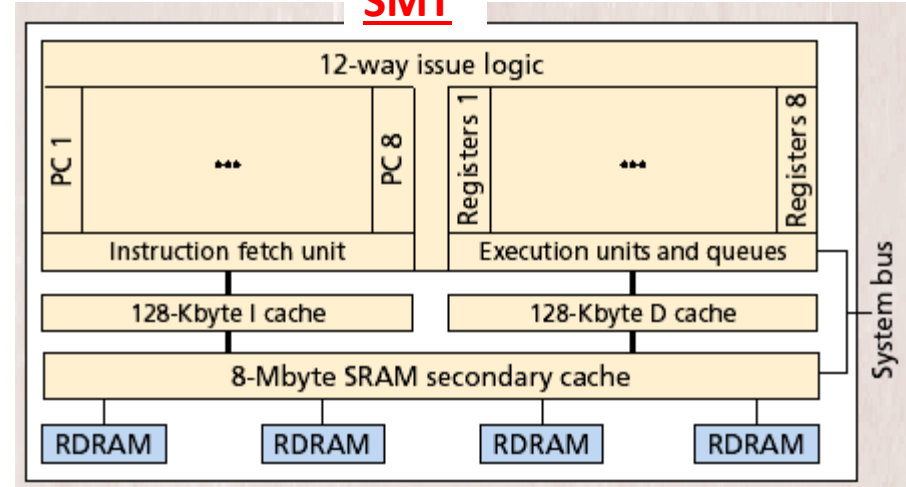


# A Single Chip Multiprocessor

## Superscalar (SS)



## SMT



# Superskalar i SMT vs. CMP

**Zašto Core-ovi:** Problemi hardverskog projektovanja (za SS i SMT):

- Površina čipa raste kvadratno sa kompleksnošću core-a
    - Broj registara  $O(\text{Instruction window size})$
    - Broj registarskih portova -  $O(\text{broj izdatih instrukcija po ciklusu})$
    - ✓ CMP rešava problem ( $\sim$  linearna zavisnost od ukupno izdatih instrukcija)
  - Sporiji takt
    - Produžavaju se žice zbog puno MUX
    - Veliki bufferi, redovi i registarski file-ovi

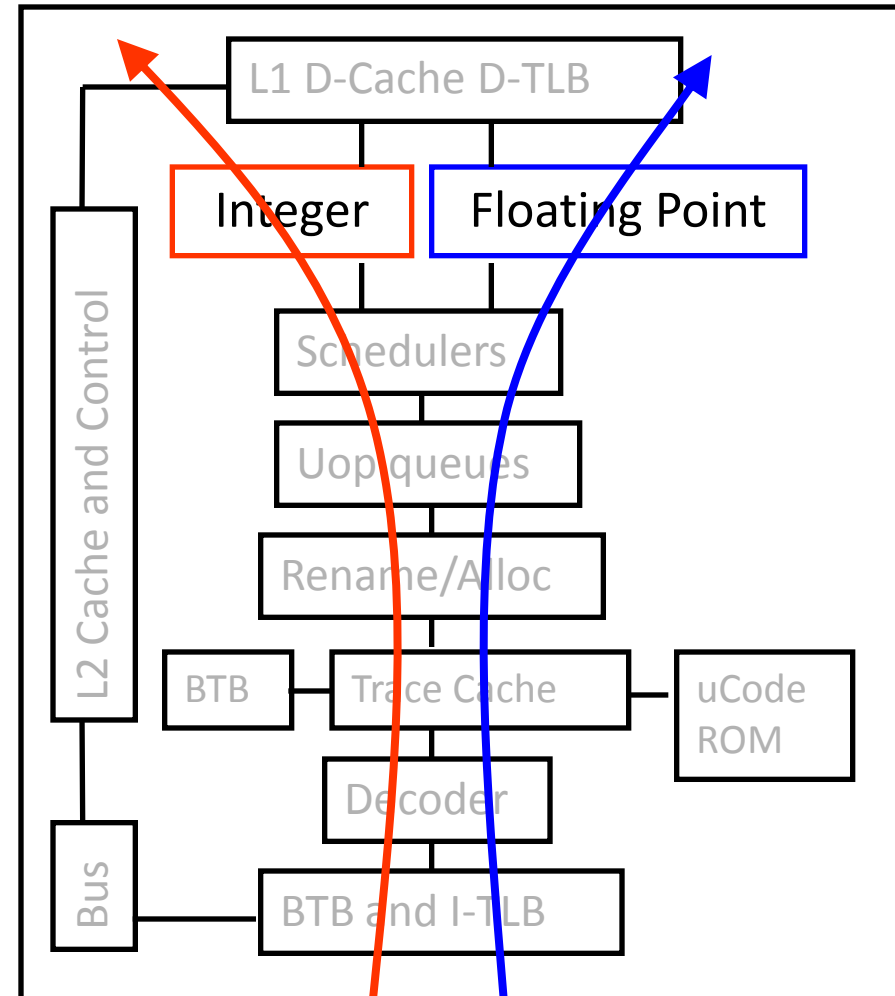
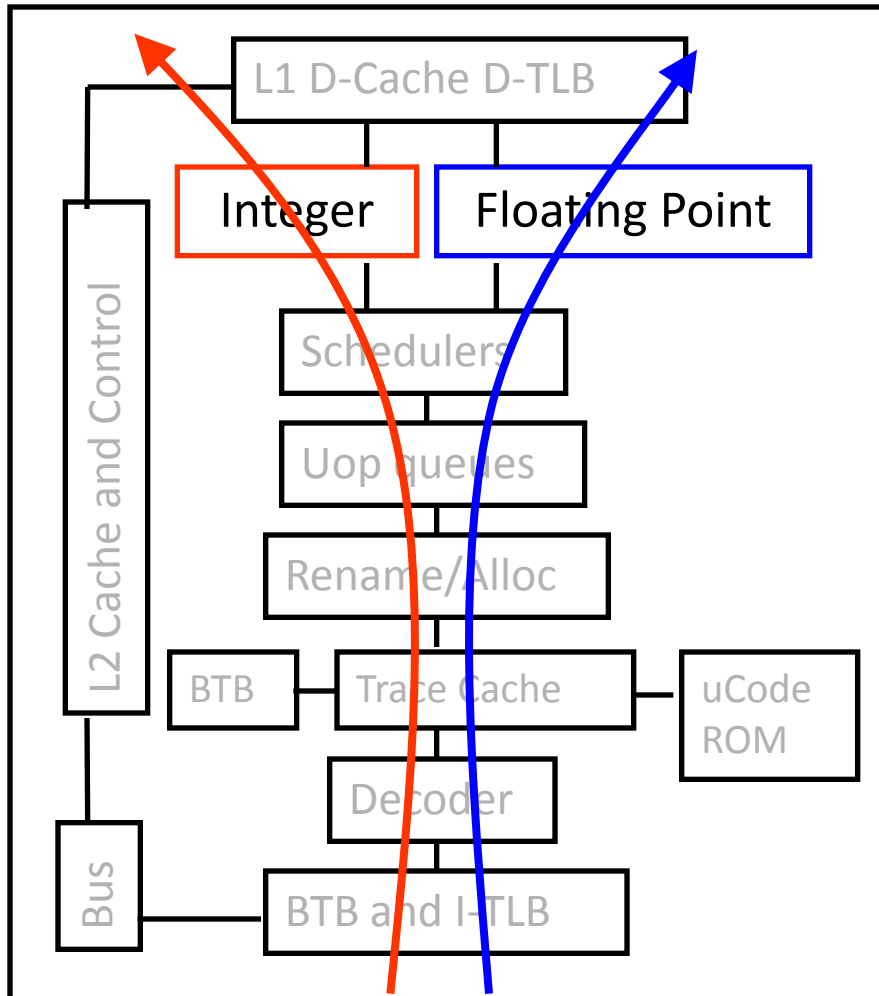
Da bi stalo na IK, smanjuje se ILP, skraćuje Pipeline, pa Branch misprediction kazne su manje

  - ✓ CMP brži takt, manji, oslanja se više na kompiler i na load balancing
- Kompleksni dizajn i verifikacija

# Budućnost

- TLP (Thread Level Paralelizam) i PLP (Process Level Paralelizam) postaju neminovnost u budućnosti i favorizuju CMP i SMT na procesorima.
- Kako iskoristiti - KDP, ali se mora obezbediti i balansiranje opterećenja procesora
- SMT (Hyperthreading) se adaptira na broj procesa i dinamički – hardverski i transparentno deli procesorske resurse prema zahtevima 2 procesa.
- Kada nema multiprocesnog rada, superskalari i SMT procesori su bolji od jednostavnih procesora u CMP
- Kompajleri i dalje igraju veliku ulogu – prevode tako da kod u instrukcijskom prozoru u svakom trenutku bude što paralelniji, ali ograničenje je nepoznavanje svih zavisnosti u vreme prevođenja

# SMT Dual-core: sve četiri niti mogu da se izvršavaju konkurentno



# 17 - 4770

- 4 Haswell core-a
- 8 thread-ova (2 po core-u)
- Niz vektorskih instrukcija
- Multiply-Add
- Enkripcija i digitalno potpisivanje u hardveru postaju deo instrukcijskog seta
- Grafičke funkcije visokog nivoa kao instrukcije – iako ima zaseban grafički procesor