# 8. Procesori koji koriste paralelizam na instrukcijskom nivou

U ovom poglavlju prikazani su procesori koji koriste instrukcijski nivo paralelizma. Oni su podeljeni u sledeće kategorije:

1. Rani RISC procesori
2. Dataflow mašine
3. VLIW mašine
4. Superskalarni procesori

Pritom se pod ranim RISC procesorima podrazumevaju oni koji su imali najviše po jednu aritmetičko-logičku jedinicu za celobrojnu aritmetiku i aritmetiku u pokretnom zarezu i kod kojih je protočnost bila jedini suštinski nivo paralelizma.

## *8.1. Rani RISC procesori*

Rani RISC procesori su svoje funkcionisanje zasnivali na nizu protočnih stepeni kroz koje prolaze instrukcije tokom različitih faza izvršavanja. Paralelizam se isključivo zasnivao na protočnosti, a redosled ulaska instrukcija u niz protočnih stepeni kompletno je bio određen redosledom definisanim u vreme prevođenja. Jedini problem vezan za zavisnost po podacima javljao se kada su u nizu susednih instrukcija, odnosno u nizu protočnih stepeni, postojale instrukcije između kojih postoji prava zavisnost po podacima. Jedan od glavnih optimizacionih zadataka programskog prevodioca bio je da prilikom optimizacije razmakne operacije (instrukcije) između kojih postoji zavisnost po podacima za dovoljno ciklusa. Tako se postiže da svi protočni stepeni rade u svakom ciklusu na izvršavanju korisnih instrukcija, naravno ako to dozvoljavaju zavisnosti po podacima.

Opišimo osnovne protočne stepene ranih RISC procesora u protočnom nizu dubine 5 stepeni:

**Dohvatanje instrukcija (Instruction fetch)**

Tokom dohvatanja, 32-bitna instrukcija preuzimana je iz keša, pri čemu su najčešće keš za instrukcije i za podatke bili odvojeni, kako bi se svakog ciklusa mogla dohvatati nova instrukcija. Kada je dohvatana instrukcija, ovi procesori radili su predikciju naredne instrukcije jednostavno dohvatajući narednu 32-bitnu instrukciju iz memorije koristeći programski brojač. Uprkos lošoj predikciji grananja (lošije od 50%), posledice pogrešne procene nisu bile velike, zbog male dubine niza protočnih stepeni. Promašaj u ovakvoj elementarnoj predikciji grananja dovodio je do "ispiranja" niza protočnih stepeni i potrebe za ponovnim punjenjem niza protočnih stepeni instrukcijama sa mesta na koje se skočilo. Na ovaj način se dakle radi spekulativno izvršavanje prvih faza instrukcija (operacija) posle instrukcije grananja, ali je instrukcija grananja započeta pre svih instrukcija predviđenih nakon grananja. Zbog očuvanog redosleda, nijedna od tih instrukcija ne završava se pre dobijanja ishoda skoka. Kao posledica, ne može se javiti problem većine izuzetaka nad spekulativnim instrukcijama (operacijama).

Kasnije verzije RISC procesora uključile su kompleksniji hardver za predikciju grananja i hardver kojim se bira adresa na koju treba da se skoči. Taj hardver doprineo je ređem

pražnjenju protočnog niza i mnogo većoj verovatnoći da će spekulativno odrađeni delovi operacija biti zaista i potrebni. Spekulativno izvršavanje po kontroli ovde se radi za sve instrukcije (operacije) koje su na izabranoj grani i koje dopunjavaju protočni niz RISC procesora do završetka instrukcije grananja.

**Dekodovanje instrukcija**

Tokom dekodovanja, generišu se dekodovani kontrolni signali i identifikuju se registri argumenata (do dva) za instrukciju koja se nalazi u ovom protočnom stepenu. Logika za izdavanje instrukcija na izvršavanje ujedno određuje da li instrukcija koja se nalazi u ovoj fazi može da ide na izvršavanje zbog zavisnosti po podacima. U suprotnom, koče se prva dva stepena u protočnom nizu. Registri argumenata istovremeno se čitaju pod uslovom da zavisnosti po podacima ne ograničavaju to čitanje.

Ako se dekoduje instrukcija koja označava bezuslovni ili uslovni skok, u oba slučaja se adresa skoka određuje u paraleli. Kod nekih procesora se u ovom stepenu već razrešava da li dolazi do skoka, čime se minimizira gubitak ciklusa u protočnom nizu u slučaju pogrešne predikcije skoka. Naravno, odgovarajući biti u statusnom registru u ovom slučaju moraju biti već određeni kao ishod operacije koja se upravo završila, pa tada mora postojati odgovarajući razmak u ciklusima između dela instrukcije skoka koji određuje signal (bit statusne reči) i dela u kome se obavlja skok. Ako je u ovom stepenu već određen ishod skoka, ciljna adresa naredne instrukcije koja se dohvata posledica je dobijenog ishoda skoka.

**Izvršavanje**

Trajanje faze izvršavanja zavisilo je od vrste operacija i mesta odakle se dohvataju argumenti. Operacije sa pokretnim zarezom trajale su više ciklusa, ali je redosled rezultata operacija sa tom vrstom argumenata bio očuvan. Operacije sa celobrojnim operandima i logičke operacije tipično su trajale jedan ciklus, mada su neke trajale dva ciklusa (npr., ako je korišćen neposredni podatak). Kako jedine zavisnosti između brojeva u pokretnom zarezu i celobrojnih postoje za operacije konverzija koje se obavljaju u jedinici za operacije u pokretnom zarezu, moralo se u hardveru voditi računa o pravilnom redosledu generisanja rezultata kod tih specifičnih operacija. Takve zavisnosti mogle su da dovedu do zaustavljanja prva dva stepena u nizu protočnih stepeni. Odvajanjem registara za celobrojne i promenljive u pokretnom zarezu su i u hardevrskoj implementaciji odvojene promenljive kod kojih se samo izuzetno javljaju zavisnosti po podacima.

**Drugi ciklus izvršavanja/NOP (access)**

U ovom delu ograničavamo se samo na operacije koje nisu operacije sa pokretnim zarezom. Kroz ovu fazu dovršava se izvršavanje, ako se radilo o operaciji koja zahteva dva ciklusa za izvršavanje. Ako je operaciji bio potreban samo jedan ciklus, rezultat se samo sprovodi kroz ovaj stepen bez izmena. Na ovaj način obezbeđeno je da ne dolazi do kolizije u upisu rezultata u registarsku memoriju i zadržava se redosled upisa rezultata. Zanemarićemo u ovom razmatranju interakcije vezane za keš za podatke. Zanimljivo je da se ovim ciklusom povećava funkcija kašnjenja zavisnosti $\delta$ samo zbog potreba zadržavanja redosleda pri upisu u nizu protočnih stepeni. Dakle, očuvanje redosleda u nizu protočnih stepeni dovodi do smanjivanja paralelizma!

**Writeback**

U ovom ciklusu upisuju se rezultati u registarsku memoriju

Uočimo da je kod ovih procesora potpuno očuvan redosled instrukcija definisanih sekvencijalno u vreme prevođenja. Taj redosled očuvan je u dohvatanju, dekodovanju, izdavanju na izvršavanje, izvršavanju i upisu rezultata. Odstupanje od redosleda upisa rezultata postoji između celobrojnih i logičkih operacija sa jedne i operacija sa pokretnim zarezom sa druge strane. Ako zanemarimo pokretni zarez, sve instrukcije (operacije) prolaze kroz svih 5 protočnih stepeni u 5 ciklusa. Odstupanje nastaje kada je neophodno zaustaviti prva dva stepena. Kod nekih RISC procesora postoji mogućnost da zadnji stepeni rade, dok su prva dva u zamrznutom stanju (stall).

Posmatrajmo sada samo one instrukcije koje nisu sa pokretnim zarezom. Na osnovu navedenog, uočava se da se instrukcije izdaju u sekvencijalnom redosledu definisanom u vreme prevođenja, u tom redosledu dohvataju se argumenti, izvršavaju operacije i vrši upis rezultata. Ovaj način rada zove se In order issue – In order completion. Naravno, zbog zavisnosti po podacima i spekulativnosti, bitan je pre svega redosled upisa (završavanja) operacije.

Jedini problem vezan za zavisnosti po podacima javlja se u ovom slučaju kada postoji prava zavisnost po podacima za operacije koje su susedne ili razmaknute za jedan stepen u protočnom nizu. Kod njih se javlja potreba da se zakoče zavisne operacije u fazi uzimanja argumenata, kako bi se pokupio rezultat upisan u fazi writeback u registre. Da bi se minimiziralo ili uklonilo ovo zaustavljanje, koristi se tehnika nazvana bypassing (forwarding).

Koristi se par (ili jedan) multiplekser koji na kraju decode stepena sprovodi do ulaza u aritmetičko-logičku jedinicu: ili rezultat operacije sa kraja četvrtog stepena – izlaza iz aritmetičko-logičke jedinice, ili sadržaj očitanog registra u decode fazi. Tako se istovremeno radi writeback i upis rezultata u registre na ulazu execute stepena protočnog niza, kada je neophodno uraditi bypassing.

Logika u decode stepenu upoređuje registre u koje se upisuje za instrukcije u execute i access stepene sa registrima koje zahteva instrukcija u decode fazi i ako postoji slaganje adresa, bira izlaz aritmetičke jedinice, a ne registar, da bude sproveden do ulaza Aritmetičko-logičke jedinice. Naravno instrukcija koja se završava ipak upisuje rezultat u registar. Na ovaj način minimizira se potreba za kompleksnim optimizacijama u vreme prevođenja i nema većih kašnjenja zbog zavisnosti po podacima. Ovde se jasno uočava razlika između Funkcije trajanja operacija d i Funkcije kašnjenja zavisnosti $\delta$, pojmova koji su bili uvedeni u modelu za List scheduling algoritam.

Da bi se rešio problem različitog trajanja operacija, a da se pritom ne koči protočni niz kada to nije potrebno, uveden je kod RISC procesora i sledeći model: In order issue – Out of order completion. Kod ovakvog modela moraju se započinjati operacije redosledom koji je definisan u vreme prevođenja, a završavaju se redosledom koji je definisan potrebnim trajanjem, ali bez nepotrebnog kašnjenja zavisnih operacija da bi se očuvao redosled generisanja rezultata. Naravno, u hardveru se tada mora obezbediti izbegavanje kolizija u korišćenju protočnih stepeni, kada se jave takve kolizije. Kolizije se mogu javiti upravo zbog različitog trajanja instrukcija (operacija). Najčešći razlozi za

uvođenje ovakvog načina rada RISC procesora su već pomenute operacije sa pokretnim zarezom, ali i operacije Load iz data keš memorije.

Osim toga, potrebno je obezbediti da se ne naruše izlazne zavisnosti i antizavisnosti, jer se redosled upisa u isti registar može izmeniti! Da bi se to postiglo, u nekim slučajevima je potrebno praviti zastoje u nizu protočnih stepeni.

## *8.2. Dataflow procesori*

Osnovna ideja Dataflow arhitektura potpuno odstupa od tradicionalne Von Neuman ili control flow arhitekture. U toj tradicionalnoj arhitekturi, iskazi u nekom programu ili pseudokodu normalno se izvršavaju sekvencijalno, u redosledu u kome su napisani. U većini programskih jezika i skupovima instrukcija procesora, pored sekvence, postoji podrška za kontrolne strukture poput uslovnog izvršavanja instrukcija (uslovni skokovi) i ponovnog izvršavanja instrukcija. Osnovna ideja Dataflow arhitektura je da se izbegnu programski brojač i programska memorija sa instrukcijama. Cilj je da se izvršavanje instrukcija (operacija) bude zasnovano samo na raspoloživosti ulaznih argumenata za instrukcije (operacije), odnosno na pravim zavisnostima po podacima. Ne postoji uspešno realizovana mašina koja se komercijalno koristi, a kako su mnogi koncepti ovih mašina preneti na superskalarne procesore, detaljnije funkcionisanje objašnjeno je u poglavlju 8.4.

## *8.3. VLIW procesori (EPIC – Explicit Parallel Instruction Computers)*

Very Long Instruction Word ili VLIW CPU arhitektura ostvaruje instrukcijski nivo paralelizma tako što postoji kontrola toka kao kod tradicionalne von Neuman arhitekture, ali se izvršavanjem grupa operacija upravlja iz veoma široke instrukcijske reči. Postoji po nekoliko protočnih izvršnih jedinica (množača, ALU, FP ALU, registarskih memorija, adresnih generatora, …) u VLIW procesoru. Broj izvršnih jedinica direktno je (eksplicitno) vidljiv u instrukcijskoj reči preko kontrolnih polja kojima se upravlja tim jedinicama. Da bi se simultano upravljalo sa tim operacijama, mora da postoji instrukcijska reč širine bar 64 bita, a postojale su implementacije sa 256 i više bita.

Osnovni koncept VLIW procesora je da kompajler odlučuje o tome koje se operacije mogu izvršavati u paraleli. Grupisanje operacija po ciklusima kompletno se planira u vreme prevođenja. Otuda se za VLIW koristio i termin grupno-sekvencijalne mašine, jer se definišu grupe operacija, a zadržava se tradicionalna kontrola toka. Kod VLIW, u okviru kompajlera mora se prilikom formiranja instrukcija obezbediti da raspored operacija poštuje zavisnosti po podacima, kao što je to bilo pokazano kod List Scheduling-a i Trace Scheduling-a.

Kontrola toka kod Trace Scheduling-a zasnivala se na činjenici da se ceo kod formira na bazi predikcije grananja u vreme prevođenja, odnosno na statički definisanim predikcijama grananja (profiling ili heuristike). Operacije sa verovatnijih grana mogu da se izvršavaju spekulativno pre grananja, a u vreme prevođenja generiše se kompenzacioni kod za odbacivanje efekata spekulativnih operacija.

Jedna od osobina VLIW arhitekture je da se hardver definiše tako da kompajler lakše obavi optimizaciju – paralelizaciju koda u vreme prevođenja.

Današnji VLIW procesor tipično je RISC-baziran i ima više od 4 glavne izvršne jedinice. Program se tipično prvo prevodi kao za RISC, a tek zatim VLIW prevodilac radi izmenu redosleda i grupisanje u tokove koji nemaju međusobne zavisnosti. Zatim se takvi tokovi pakuju u susedne velike instrukcijske reči. Takav način generisanja koda omogućava kompatibilnost koda i mogućnost da se generišu različita prepakivanja za VLIW sa različitim nivoom paralelizma, a da se ujedno obezbedi kompatibilnost prema RISC mašinama. Inače, nekompatibilnost na nivou mašinskih instrukcija, dugo je smatrana jednom od glavnih mana ranih VLIW mašina.

Danas je najpoznatiji VLIW procesor Intel - HP Itanium IA-64 EPIC procesor. Osim toga, gotovo svi procesori za digitalnu obradu signala su VLIW (npr. Texas Instruments)

## *8.4. Superskalarni procesori*

Osnovna ideja superskalarnih procesora je da se u paraleli dohvata veći broj skalarnih instrukcija i unosi u procesor u paketima od 2-8 instrukcija po ciklusu. Prilikom tog dohvatanja, instrukcije se preuzimaju iz interne instrukcijske keš memorije koja ima reč od 128 ili više bita širine. Za razliku od VLIW, ovde su kompletne, tipično 32-bitne, RISC instrukcije napakovane u široku reč instrukcijskog keša, kako bi u svakom ciklusu mogle da se pročitaju najmanje 2 instrukcije. Zato će se u opisu koristiti termin instrukcija, a ne operacija, mada one imaju isto značenje.

U daljem tekstu će biti pretpostavljeno da se dohvataju po četiri 32-bitne instrukcije iz instrukcijskog keša istovremeno. Sve te instrukcije dekoduju se u paraleli, tako da na izlazu logike za dekodovanje imamo kompletnu informaciju o vrstama instrukcija koje su upakovane zajedno u reč instrukcijskog keša. Za početak, pretpostavićemo da nijedna nije instrukcija skoka. Tada sve instrukcije postaju kandidati za izvršavanje. Do ovog trenutka, sve instrukcije imale su redosled definisan u vreme prevođenja. Međutim, superskalarni procesor sada prima dekodovane instrukcije u nešto što će biti definisano apstrakcijom zvanom instrukcijski prozor. Cilj instrukcijskog prozora je da za sve operacije u instrukcijskom prozoru generiše dinamički graf zavisnosti po podacima koji bi uključivao i prave zavisnosti, ali i antizavisnosti i izlazne zavisnosti, ako se ne uradi preimenovanje. Kasnije će biti pokazano da se uvek radi dinamičko preimenovanje.

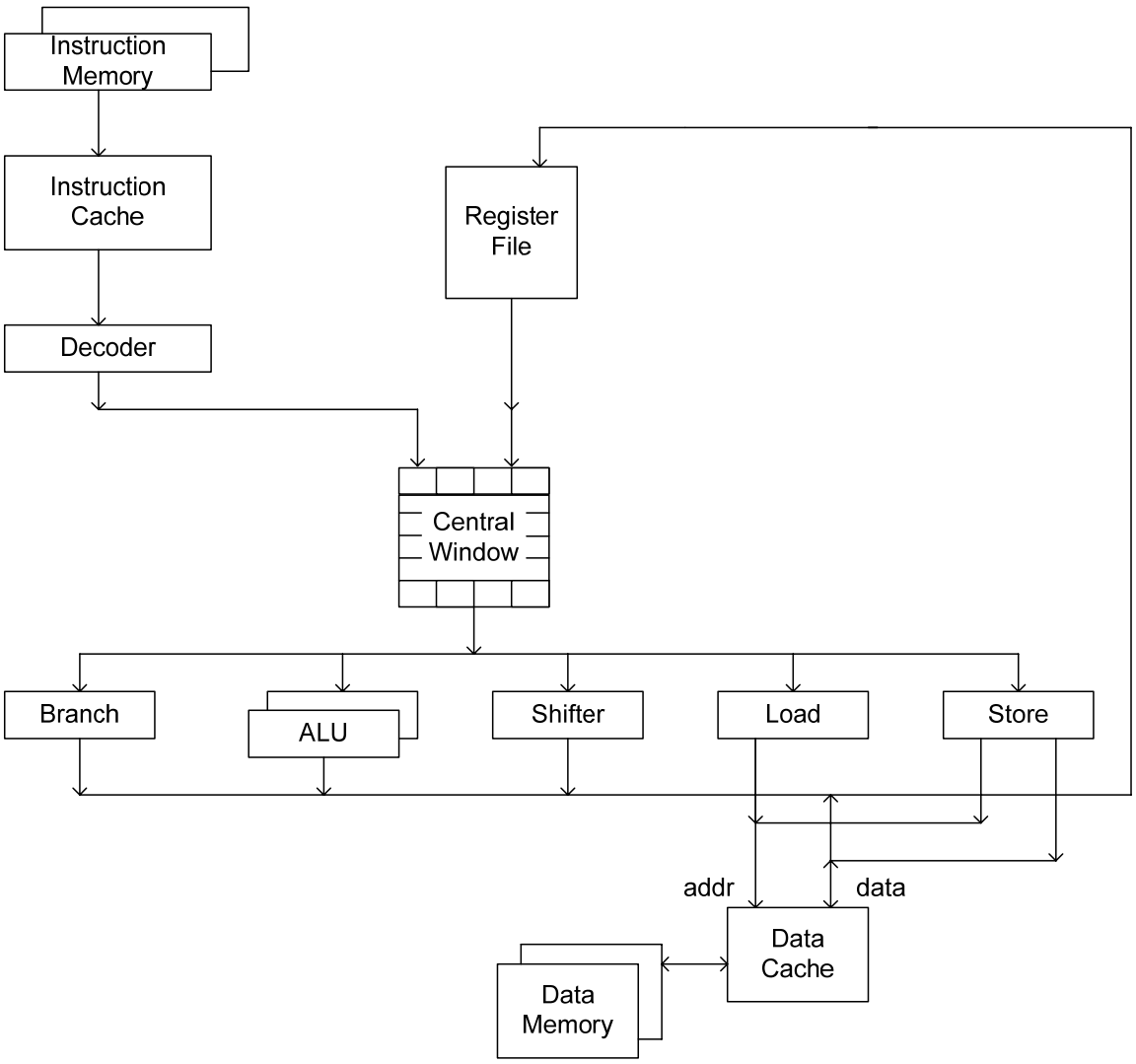
Na osnovu takvog dinamičkog grafa, sada sve instrukcije unutar instrukcijskog prozora mogu da se klasifikuju u data ready i zavisne instrukcije. Svaka zavisna instrukcija čeka da se izmeni graf nakon završetka instrukcija u protočnim stepenima za izvršavanje, da (oba) operand(a) postane(u) data ready. Kada svi operandi operacije postanu izračunati, ona postaje data ready. Naravno, slanje instrukcija na izvršavanje oslobađa mesta u instrukcijskom prozoru i novi paketi dekodovanih instrukcija mogu se učitati u instrukcijski prozor.

U apstrakciji sa instrukcijskim prozorom, popunjavanje instrukcijskog prozora obavlja se in-order – po redosledu, ali se izdavanje instrukcije na izvršavanje i upis rezultata radi out of order – izvan redosleda (iz vremena prevođenja). Dakle ovde je u pitanju out of order issue i out of order completion. Pritom zavisnosti po podacima određuju kada

instrukcija (operacija) postaje data ready i kada ide (se izdaje) u izvršavanje, ako ima dovoljno aritmetičko-logičkih jedinica. Broj stepeni izvršnih jedinica određuje trajanje izvršavanje instrukcije. Naravno, broj operacija koje mogu da idu u izvršavanje po ciklusu treba da bude približno jednak broju instrukcija (operacija) koje se po ciklusu mogu ubaciti u instrukcijski prozor.

U ovoj predstavi, bitno je uočiti da instrukcijski prozor funkcioniše tako da odvaja sinhrone protočne stepene dohvatanja paketa instrukcija i njihovog paralelnog dekodovanja od kasnijeg izvršavanja. U svemu tome instrukcijski prozor ima ulogu veoma specifične elastične memorije.

Naravno da je prethodna apstrakcija sa instrukcijskim prozorom pojednostavljenje, kako bi se opisao osnovni princip rada superskalarnog procesora. Prvo će se detaljnije razmotriti način popunjavanja instrukcijskog prozora. Kasnije će detaljnije biti razmatran način određivanja zavisnosti po podacima u instrukcijskom prozoru, a na kraju šta je zaista registarski fajl i kako se rešava problem preimenovanja zbog eliminacije antizavisnosti i izlaznih zavisnosti.




Sl. 8.1. Uprošćena slika superskalarnog procesora sa apstrakcijom centralnog prozora

### 8.4.1. Uslovni skokovi i popunjavanje instrukcijskog prozora

Uslovni skokovi očigledno predstavljaju mesto na kome se ne može sa sigurnošću odlučiti odakle popunjavati instrukcijski prozor. Zato se radi dinamička predikcija grananja (spekulativno izvršavanje po kontroli). Ona se obavlja tako što se u paralelnom dekodovanju detektuju instrukcije skoka. Za prvu instrukciju skoka iz paketa (reči instrukcijskog keša) radi se predikcija grananja i kada se ne predviđa skok, sve instrukcije iz paketa unose se u instrukcijski prozor, ako nema još jednog grananja u istom paketu. Kada se predvidelo da će doći do skoka, instrukcije iza instrukcije skoka ne unose se u instrukcijski prozor. Ako postoji još grananja u istom paketu, prethodna logika se ponavlja za taj dodatni skok.

Ako se predvidi skok, doskače se na mesto gde se dohvata novi paket (reč instrukcijskog keša) od 4 instrukcije. Naravno, adresa instrukcije na koju se skače ne mora da bude na početku paketa instrukcija, već može da se nalazi na bilo kom mestu instrukcija u paketu (reči instrukcijskog keša).

Na slici 8.1. prikazan je slučaj da 4 originalne instrukcije ulaze u reč instrukcijskog keša, što je najčešći slučaj kod današnjih procesora. Treba napomenuti da je unos instrukcije skoka u instrukcijski prozor više apstrakcija nego realnost u hardveru. U suštini, instrukcija skoka obrađuje se u prediktoru grananja i logici koja utvrđuje da li je došlo do izuzetka zbog pogrešne predikcije grananja. Jedini razlog zašto se vezuje za instrukcijski prozor su činjenice da se nekada izračunava adresa i da se signal (bit paralelne procesorske statusne reči) najčešće dobija sa izlaza neke aritmetičke jedinice. Ako nisu potvrđeni ishodi prethodnog grananja, predviđeni ishodi skoka uzimaju se kao vrednost za predikcije sledećih grananja na dinamičkom tragu. Dakle, u svim osnovnim elementima logike rada, dinamički prediktor radi kako je opisano u poglavlju o spekulativnom izvršavanju.

Ako se popunjavanje radi tako što se predviđa da neće biti skoka, onda se sve 4 instrukcije ubacuju u instrukcijski prozor. Instrukcije grananja čekaju da kasnije dobiju izračunati flag (bit procesorske statusne reči) za skok, kako bi prediktor razrešio da li je pogodio stvarni ishod grananja. Ukoliko se predvidi skok, instrukcije koje su prethodile skoku ubacuju se u instrukcijski prozor zajedno sa instrukcijom skoka, a instrukcije posle instrukcije skoka u "paketu" od 4 instrukcije ne ubacuju se u instrukcijski prozor. Skok, sa stanovišta dekodovanja pre instrukcijskog prozora, obavlja se na kvante od 4 instrukcije (početke 128-bitnih reči sa paketima od 4 32-bitne instrukcije), uz zadržavanje evidencije o pravoj adresi skoka na nivou pojedinačne instrukcije. To je naravno moguće, jer se adresa skoka odnosi na pojedinu instrukciju, a ona ujedno jednoznačno određuje adresu paketa od četiri instrukcije u kojoj se nalazi instrukcija na koju se skače.

Instrukcije učitane u paketu nakon skoka, a koje su prethodile instrukciji na koju se stvarno skače, ne ubacuju se u instrukcijski prozor. Dakle, instrukcije skoka i adresa na koju se skače unutar paketa instrukcija predstavljaju granice za stvarno učitavanje instrukcija iz paketa u prozor. To važi i za paket iz koga se skače i za paket na koji se

skače. Ako se iz instrukcijskog keša učitavaju 4 instrukcije maksimalno po ciklusu, neposredno pre skoka ili na adresi skoka, u instrukcijski prozor može se upisivati od 1 do 4 instrukcije u tom ciklusu.

Na ovom mestu neće se razmatrati šta se događa u slučaju greške u predikciji, jer je ta tema kasnije obrađena.

### 8.4.2. Određivanje zavisnosti i preimenovanja

Jedino mesto na kome se može lako odrediti zavisnost po podacima je mesto na kome su instrukcije još u originalnom redosledu. Dakle, to je faza na izlazu iz paralelnog dekodera. Tada se komparacijom argumenata i rezultata u dekodovanim instrukcijama i poznatog redosleda paketa instrukcija može odrediti zavisnost po podacima. Međutim, čak unutar jednog paketa može se dogoditi da postoje dva upisa u isti registar i jedno čitanje između! Tada naravno kôd registra u instrukciji ne obezbeđuje jednoznačno definisanje zavisnosti. Osim toga, čak i kada bi se jednoznačno definisale zavisnosti, zaostale bi nepotrebne izlazne zavisnosti i antizavisnosti. Naravno, u slučaju više paketa instrukcija koji ulaze u prozor, problem postaje još veći.

Rešenje za oba problema nađeno je u preimenovanju registara svaki put kada se vrši upis. Zato, umesto jednog registra Ri, prema kodu instrukcije, može se javiti nekoliko registara koji su preimenovane verzije registra Ri unutar instrukcijskog prozora. Time se eliminišu antizavisnosti i izlazne zavisnosti, ali i definišu jedinstvene vrednosti za koje mogu lakše da se vežu zavisnosti!

Uvedimo sada uobičajene termine. Definišimo prvo šta su Arhitekturalni, a šta Fizički registri. Programi u mašinskom jeziku definišu čitanja i upise za ograničeni broj registara definisanih arhitekturom skupa instrukcija. Taj broj registara tipično je stepen broja 2 i u instrukcijama postoje polja od nekoliko bita koji određuju "vidljive" registre. Oni se nazivaju **arhitekturalnim registrima.** Izvršni program će uvek referisati te registre, i tipično se ispitivanje programa zaustavljanjem pomoću debugger-a na breakpoint-u ili watchpoint-u svodi na čitanje vrednosti arhitekturalnih registara. Tako se programeru prikazuje stanje kao da se program izvršavao u sekvencijalnom redosledu i zaustavio tačno tamo gde je odredio programer koji testira kôd. Pritom se prikazuju vrednosti arhitekturalnih registara u tom trenutku sekvencijalnog izvršavanja.

Broj fizičkih registara mora da bude veći, jer svakom arhitekturalnom registru mora da bude pridružen bar jedan fizički registar. Kad god se radi preimenovanje, novi fizički registar mora se alocirati, pa je potreban broj fizičkih registara bar za 2-3 puta veći nego broj arhitekturalnih. Pritom treba na neki način sačuvati sva preslikavanja arhitekturalni registar – fizički registar. Sva preimenovanja zasnivaju se na sledećem principu. Svaka referenca arhitekturalnih registara (upis ili čitanje) konvertuje se u tagove instrukcija (operacija). Tagovi imaju 2-3 bita više od adresa arhitekturalnih registara, zbog potrebe da se adresira više fizičkih registara. Logika preimenovanja pomoću tagova je sledeća:

1. upis rezultata instrukcije u registar dovodi do formiranja novog tag-a
2. svako čitanje, do novog in order upisa u taj arhitekturalni registar dobija isti tag.

Pritom se moraju paralelno raditi preimenovanja za adrese rezultata svih instrukcija koje ulaze u instrukcijski prozor, ali odmah i preimenovanje argumenata svih tih instrukcija u skladu sa pravilom iz prethodnog pasusa. Dakle, Rename file u našem primeru sa procesorom koji ima 4 instrukcije u paketu (reči instrukcijskog keša) mora da ima 4

write porta za upis novog preslikavanja adresa arhitekturalnog registra rezultata – adresa fizičkog registra. Često je to preslikavanje adresa arhitekturalnog registra - adresa arhitekturalnog registra + 2-3 dodatna bita, što čini tag. On takođe mora da ima i 8 read portova za priduživanje tagova argumentima te četiri instrukcije koje treba da uđu u instrukcijski prozor svakog ciklusa. Ova logika rename file-a kao multiportnog registarskog file-a veoma je kompleksna, a kompleksnost raste kvadratno sa brojem portova.

Sada je očigledno da tag jedinstveno definiše reference unutar instrukcijskog prozora. Zato komparacija vrednosti tag-ova direktno govori o tome da li postoje zavisnosti po podacima. U različitim šemama, različito se određuju zavisnosti, ali najčešći princip je sledeći:

1. U trenutku kada se instrukcija ubacuje u instrukcijski prozor, argumenti dobijaju tagove koji odgovaraju tagu generisanom kada je ubacivana instrukcija koja generiše potreban rezultat, ali i bit koji govori da li je izračunat rezultat za taj tag. Taj bit naziva se data ready bit argumenta i odmah može imati vrednost ready, mada je češće not ready. (Ne razmatra se slučaj da u istom paketu generišemo novi tag i imamo operaciju koja koristi novoformirani tag – jasno je da tada data ready tag argumenta ima vrednost data ready bita – not ready)
2. Ako nije izračunat rezultat za taj tag, argument se proglašava da nije data ready i dobija takvu vrednost pridruženog data ready bita.
3. Kada neka instrukcija izračuna rezultat za taj tag, dodati bit za zavisnost po podacima svih argumenata svih instrukcija u instrukcijskom prozoru označenih tim tagom dobijaju vrednost data ready. Ovo je asocijativni upis. Na ovaj način, iz dinamičkog grafa zavisnosti po podacima za instrukcijski prozor nestaje izračunata instrukcija.
4. Pretpostavimo instrukciju sa dva argumenta. Tada oba argumenta moraju da budu data ready da bi instrukcija postala data ready i mogla da ide u izvršavanje.

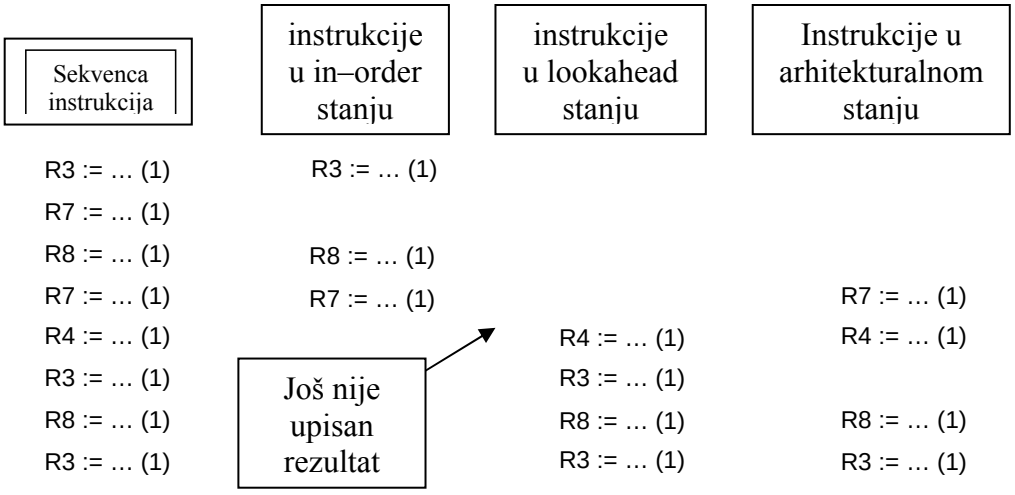
Na ovaj način se preko tagova realizuje određivanje zavisnosti po podacima u instrukcijskom prozoru. Zavisnosti po podacima otkriju se preko uparenih vrednosti tagova u instrukcijskom prozoru, a beleži se preko data ready bita argumenata (aktuelno stanje zavisnosti argumenta u dinamičkom grafu). Da bi se lako radilo uparivanje vrednosti tagova, očigledno je da će morati da postoji paralelna asocijativna memorija koja će npr. za sve lokacije kod kojih postoji isti tag na bilo kom argumentu generisati paralelni upis u data ready bit pronađenog upravo izračunatog argumenta. Memorija se pritom naziva paralelnom asocijativnom, zato što se to mora raditi u paraleli za sve izračunate rezultate.

Zanimljivo je i kako se određuje da li se neki fizički registar sa pridruženim tagom može prepisati. Ako postoji ponovni upis u njemu pridruženom arhitekturalnom registru sa formiranim novim tag-om, a sve ostale operacije iz instrukcijskog prozora koje su trebalo da korise taj tag su već pročitale argumente, registar se može ponovo koristiti za novo mapiranje bilo kog arhitekturalnog registra u taj tag. Time je definisana logika oslobađanja fizičkih registara za nova preimenovanja i određivanja zavisnosti po podacima.

### 8.4.3. Stanja instrukcija

Instrukcije koje ulaze u instrukcijski prozor u tipičnom slučaju su spekulativne operacije po kontroli. Najčešće su one još uvek spekulativne i u trenutku kada generišu rezultat.

Zato je neophodno jasno definisati stanja u kojima instrukcija može da se nalazi. Tek kada su sve instrukcije koje su joj prethodile završene i upisale rezultat, rezultat instrukcije sigurno prestaje da bude spekulativan i tada se radi ekvivalent commit operacije. Stanje do zadnje kompletirane instrukcije tada se tretira kao ekvivalent sekvencijalnog izvršavanja. Kako je sa stanovišta commit operacije svejedno da li se radi o instrukciji uslovnog grananja ili drugoj instrukciji, ovim se postiže da one operacije koje su doživele commit sigurno nisu spekulativne. Sa druge strane, operacije iz istog bazičnog bloka kao neke koje su već doživele commit sigurno nisu spekulativne, ako nema izuzetaka. Kako je ista logika korišćena za izuzetke i grešku u predikciji grananja, neophodno je da sve operacije pre neke operacije dožive commit, da bi ta operacija mogla da doživi commit. Naravno commit mora da se radi paralelno za više operacija ugradnjom paralelne logike u procesor.




Sl. 8.2. Stanja instrukcija u izvršavanju

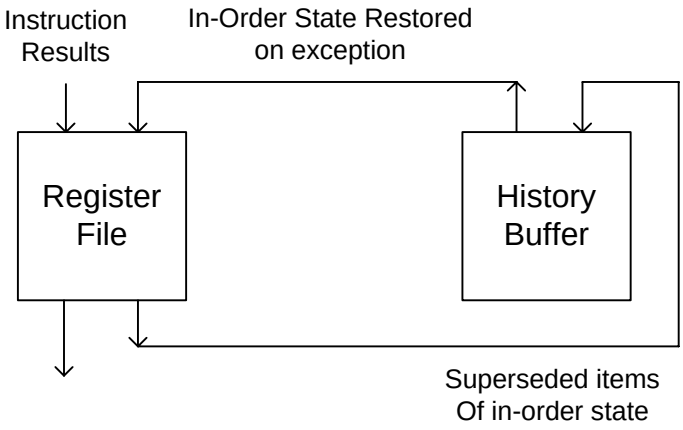
Stanja instrukcija u ovom kontekstu definisana su na sledeći način:

1. In-order – u osnovi instrukcije koje su doživele commit. Njihov rezultat može se još koristiti od strane drugih instrukcija. One su definisale stanje arhitekturalnih registara za poslednje vrednosti arhitekturalnih registara koje su doživele commit. Za pamćenje in-order stanja registara procesora potrebno je onoliko registara koliko imamo arhitekturalnih registara.
2. Lookahead – sve instrukcije koje nisu doživele commit svojih rezultata. Tu pripadaju sve instrukcije nakon trenutne tačke poslednjeg commit-a. Velika većina su spekulativne operacije, ali neke ne moraju da budu (ako zanemarimo izuzetke), jer jednostavno nisu završene operacije iz bazičnog bloka nakon poslednjeg grananja koje je doživelo commit, a pre sledećeg grananja za koje nemamo potvrdu da je predikcija bila dobra. Kako se rade preimenovanja i dodeljuju odgovarajući tagovi, instrukcija u ovom stanju ima više nego u in-order stanju.
3. Arhitekturalno stanje koje sadrži poslednje upisane vrednosti u fizičke registre koji odgovaraju arhitekturalnim registrima, bez obzira da li je u posmatranom trenutku in-order ili lookahead.

To je ilustrovano primerom na Sl. 8.2. Za paralelni commit koristi se reorder buffer koji ima zadatak da prima sve rezultate instrukcija i da radeći paralelnu analizu grupe instrukcija paralelno prebacuje iz lookahead stanja u in-order stanje. Otuda i naziv koji asocira na vraćanje u normalan redosled nakon paralelizacije. Dakle, u njemu se čuvaju informacije o tome da li su stigli rezultati instrukcija (i time ažuriraju data ready biti argumenata), ali je interni redosled instrukcija onaj kojim su došle u instrukcijski prozor. Tako se lako pokazuje na instrukcije koje su doživele commit. Istovremeno, generišu se paralelni upisi u skup registara koji definiše i in-order stanje arhitekturalnih registara, ali i lookahead stanje korišćenjem ranije opisanih tag-ova. Za nekoga ko želi da pročita spolja arhitekturalne register, reorder buffer stvara iluziju da su instrukcije izvršene sekvencijalno.

### 8.4.4. Oporavak od izuzetaka i greške u predikciji grananja

U superskalarnim procesorima, upravo zahvaljujući ideji da se commit ne vezuje samo za instrukcije grananja, moguće je jedinstveno tretirati problem greške u predikciji grananja i pojavu izuzetaka. Bitno je da se nekako procesor može vratiti u stanje za koje je postignut commit. To znači da prilikom oporavka od izuzetaka (greške u predikciji grananja) postoje samo in-order stanja i po jedna upisana vrednost za svaki arhitekturalni registar. Ključnu ulogu očigledno treba da odigra reorder buffer. Najjednostavnija šema oporavka prikazana je na Sl. 8.3.



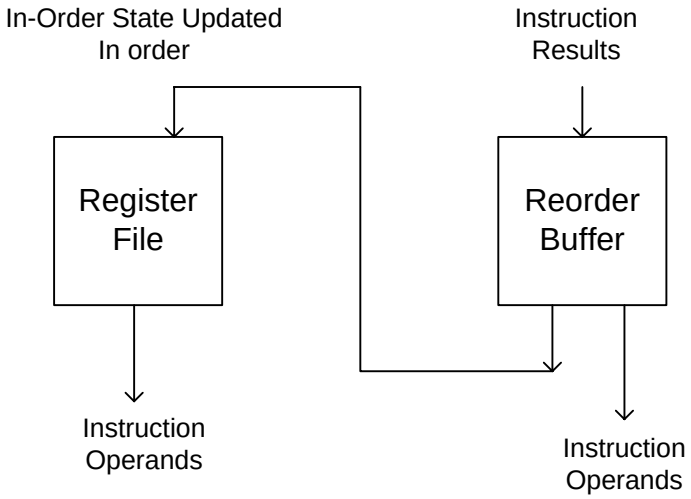

Sl. 8.3. History buffer

Šema pokazuje da se u toku izvršavanja ažurira **history buffer,** tako da on pokušava da drži ažurno in-order stanje. Ova šema ima dve ozbiljne mane u implementaciji. Traže se dodatni portovi na Register file-u za prebacivanje novih in-order vrednosti u history buffer. Zapamtimo da takvih portova treba da bude više (npr. najmanje 4 u našem slučaju). Osim toga, pri oporavku od exception-a ili greške u predikciji grananja, mora da potraje vraćanje stanja arhitekturalnih in-order registara u register file, jer postoji ograničenje u broju portova. Naravno, register file je veći od history buffer-a po broju lokacija, zbog preimenovanja arhitekturalnih registara i činjenice da on mora da čuva tag-ove. Rešenje sa history buffer-om ima i jednu prednost – potpuno je odvojeno razrešavanje izuzetaka od posla alokacije i dealokacije tag-ova i registara u registarskom file-u, kao i od određivanja zavisnosti po podacima.

Da bi se otklonile navedene mane i smanjio broj portova, uvedeno je rešenje u kome **reorder buffer** čuva i in-order i lookahead stanje i radi update in-order stanja u registarskom file-u. U ovom slučaju registarski file odgovara in-order stanju. Pritom se

pretpostavlja da reorder buffer uključuje registarski file koji sadrži sve izračunate vrednosti rezultata, dakle in-order i lookahead stanje, pored celokupne logike određivanja zavisnosti po podacima i in order stanja. Logika ažuriranja registarskog file-a identična je kao kod History buffer-a. Međutim, operandi instrukcija mogu se čitati iz reorder buffer-a i iz register file-a. Pri tome, operandi iz registarskog file-a koriste se u slučajevima kada je rezultat in-order stanja još uvek potreban nekoj operaciji kao argument, čak i kada nema izuzetaka. Ovakvom logikom mogu se ranije oslobađati registri iz reorder buffer-a, za operacije koje su izgenerisale in order rezultate.

Reorder buffer napravljen je kao specifičan FIFO, u kome se alociraju mesta za instrukcije po redosledu ulaska u instrukcijski prozor. Rezultati se upisuju u te rezervisane lokacije nakon završetka instrukcije. Tako se popunjavaju lokacije u FIFO memoriji, a prebacivanjem svih koji su doživeli commit u register file, postiže se ažurno in-order stanje.

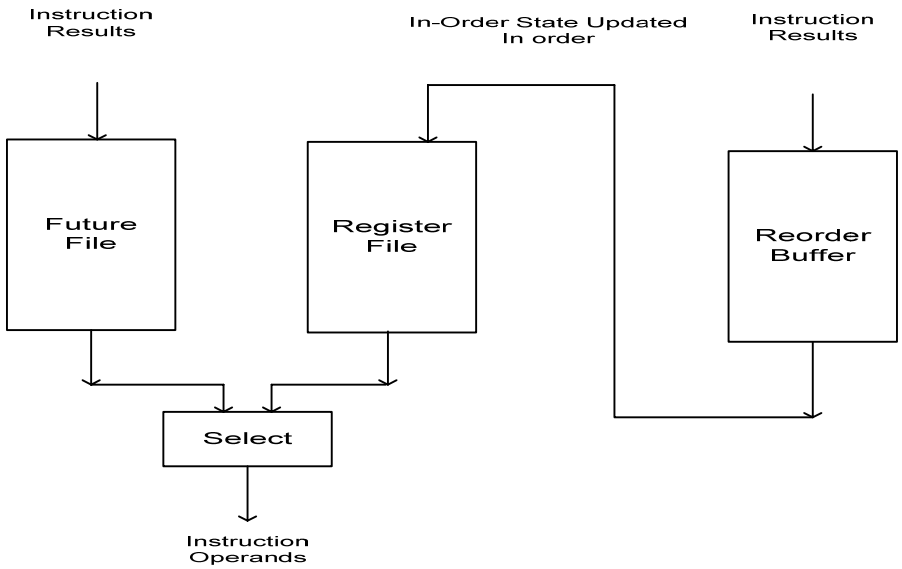



Sl. 8.5. Rešenje sa reorder buffer-om i retirement register file, odnosno register file-om

Ovim rešenjem postiže se da nakon izuzetka može odmah da krene izvršavanje na bazi sadržaja register file-a, jer je dostupan za direktno prebacivanje argumenata do operacija u izvršavanju. Tako se skraćuje vreme oporavka od izuzetaka. Uzimanjem argumenata (operanada) instrukcija sa oba mesta smanjuje se ukupan broj portova pojedinačnih registarskih memorija u odnosu na prethodno rešenje. Mana je donekle potreba za asocijativnim pretraživanjem reorder buffer-a zbog kombinovanja in-order i lookahead stanja, ali je ta funkcionalnost već morala da se obezbedi za reorder buffer zbog dinamičkog određivanja zavisnosti po podacima.

Rešenje koje dodatno ubrzava rad i oporavak je rešenje sa **future file**-om. Kod njega se u future file-u čuva arhitekturalno stanje, a reorder buffer ima ulogu samo da ažurira in order stanje. Rezultati se upisuju istovremeno u Future file i u reorder buffer. Register file čuva in order stanje. Ključna prednost je u ukidanju potrebe da se pravi asocijativno pretraživanje prilikom čitanja argumenta. U nekim izvedbama, ovo rešenje ograničava broj preimenovanja. Naravno, raste i broj lokacija registara, ali uz smanjenje prosečnog broja portova svake registarske memorije.

Kao rekapitulacija, instrukcije obave commit po redosledu, primaju se po redosledu u procesor (instrukcijski prozor), ali se šalju na izvršavanje, izvršavaju i generišu rezultate izvan redosleda. Za nekog spolja, ulazni i izlazni tok su na kraju kao po redosledu, a celokupna kompleksna logika za paralelizaciju u toku izvršavanja je skrivena.




Sl. 8.6. Rešenje sa Future file-om (Active Register File)

Očuvanje in-order završavanja postiže se postavljanjem reorder buffera neposredno ispred registarskog file-a koji treba da čuva in-order stanje. Pomeranje instrukcijskog prozora se u osnovi obavlja tako što reorder buffer čuva sve instrukcije i rezultate do prve sledeće koja treba da doživi commit, a na osnovu originalnog FIFO redosleda formiranog prilikom učitavanja instrukcija u procesor.

### 8.4.5. Varijante realizacije preimenovanja

Postoje dve osnovne varijante realizacije preimenovanja. Kod **tag-indexed register file** varijante, postoji samo jedan veliki registarski file za podatke koji se adresiraju pomoću tagova. Kada se instrukcija predaje izvršnoj jedinici, generišu se tagovi i šalju do fizičkih registara koji onda generišu argumente.

Druga varijanta – **rezervacione stanice** podrazumeva postojanje većeg broja asocijativnih registarskih file-ova na ulazima izvršnih jedinica. Svaki operand svake od instrukcija u redu čekanja na izvršavanje neke izvršne jedinice ima mesto u jednom od ovih registarskih file-ova. Kada se u ovom slučaju instrukcija predaje izvršnoj jedinici, predaju se i vrednosti iz registarskog file-a. Najčešće se rezervacione stanice prave za sledeće grupe izvršnih jedinica: integer, floating point, load/store, branch i address unit.

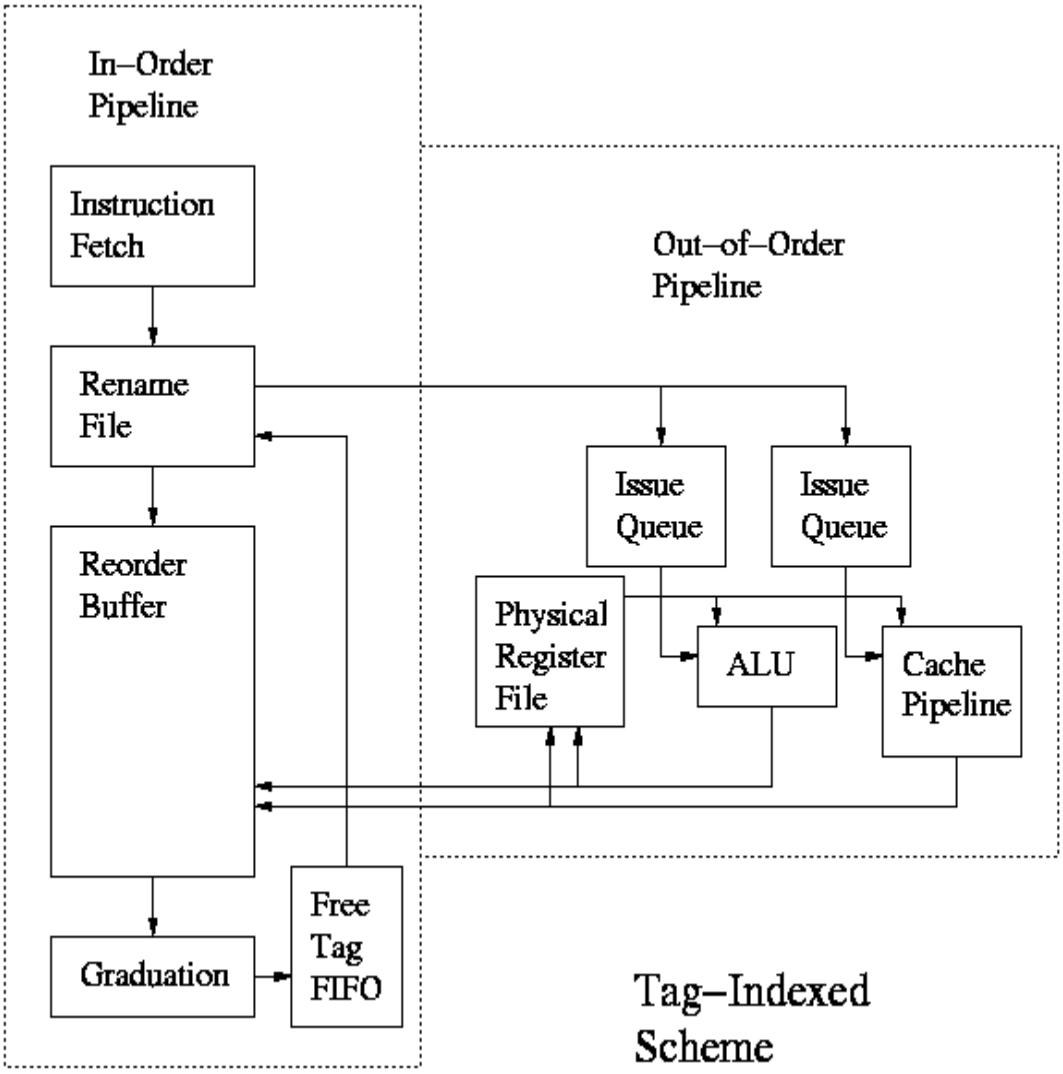
8.4.5.1. Preimenovanje sa tag-indexed register file

U fazi preimenovanja, svaka referenca arhitekturalnog registra preslikana je u arhitekturalno indeksiranom **Remap (ili Rename) file-u**. Ovaj file generiše novi tag i ready bit za budući rezultat instrukcije, a ready bit je uvek 0. Istovremeno, generišu se tag-ovi za argumente operacije i odgovarajuće vrednosti ready bita uz tagove

argumenata. Tag argumenta nije ready ako postoji instrukcija sa tim tag-om u instrukcijskom prozoru koja će tek raditi upis u taj tag-indeksiran registar, jer nije još završena. Za čitanje, tag postaje adresa sa koje treba da se čitaju argumenti instrukcija.

Za svaki novi upis, novi tag uzima se iz skupa slobodnih tagova, a novo preslikavanje arhitekturalnog registra u fizički registar upisuje se u remap file. Prethodno alocirani fizički registar, za taj arhitekturalni registar, i dalje se pamti, ukoliko je potreban. U **reorder buffer-u**, koji u FIFO redosledu čuva instrukcije u originalnom programskom redosledu od dekodovanja, zadržava se referenca prema odgovarajućoj instanci tag-a za upis, do **istovremenog** ispunjenja sledećih uslova: commit rezultata (in order), postoji bar jedan noviji tag za isti arhitekturalni registar, sve reference čitanja argumenata tog tag-a od strane instrukcija u instrukcijskom prozoru su kompletirane.

Sve instrukcije tretiraju se od tog trenutka kao da su u redovima za izvršavanje. Da li instrukcija u takvim redovima zaista može da ide u izvršavanje, zavisi od vrednosti pridruženog ready bita za svaki argument.

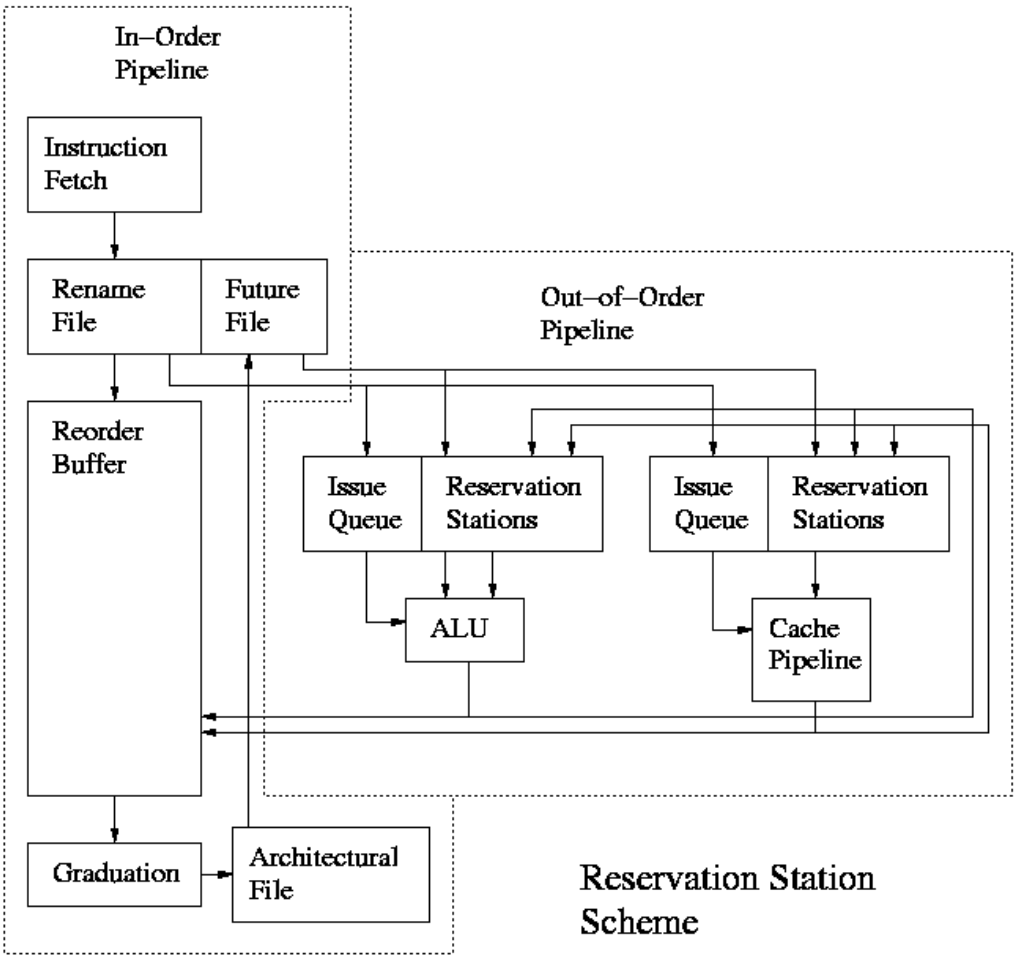



Exception ili branch misprediction dovodi do vraćanja rename (remap) stanja na poslednju validnu instrukciju.

Svaka izvršena instrukcija emituje vrednost tag-a tako da instrukcije u redovima za izvršavanje mogu da uporede ove emitovane tagove sa tagovima sopstvenih nespremnih argumenata. Ako postoji jednakost tagova emitovanog taga i tag-a argumenta, to znači da je operand spreman i menja se njegov ready bit. Ovu vrednost prima i remap file, tako da nove instrukcije koje referišu taj tag dobijaju da je taj argument ready. Data ready svih operanada znači da je instrukcija spremna za izvršavanje. Redovi za izvršavanje tada šalju instrukcije do različitih funkcionalnih jedinica. Nespremne instrukcije ostaju da čekaju u redu za izdavanje instrukcija i time se realizuje centralni prozor. Sazrevanje - commit nekog registra izaziva automatski zastarevanje prethodne vrednosti tog arhitekturalnog registra, čime se obezbeđuje jedan od potrebnih preduslova da se oslobodi lokacija za nova preslikavanja i time oslobodi tag. Kompletna logika sazrevanja (graduation) dovodi do popunjavanja FIFO-a sa slobodnim tag-ovima, tako da se gotovo nikada ne desi da nedostaju slobodni tag-ovi.

8.4.5.2. Preimenovanje sa rezervacionim stanicama

Razlika u odnosu na prethodnu realizaciju je da se arhitekturalni registri za očitavanje odmah istovremeno traže u rename file-u i future file-u. Future file sadrži vrednost spremnog argumenta, ako nema upisa novih vrednosti u taj arhitekturalni registar od strane instrukcija u izvršavanju. U tom slučaju, argument je odmah ready. Tada se instrukcija prebaci u issue red, a odmah i argument prebaci kao ready u rezervacionu stanicu. Tagovi se sekvencijalno dodeljuju kod pojave svake instrukcije koja upisuje po logici kružnog dodeljivanja.




Reservation Station Scheme

U issue queue svake jedinice instrukcije čekaju na operande da postanu ready. Svaki emitovani rezultat zajedno sa tagovima upisuje se još i u rezervacione stanice, tamo gde je potreban. Izdate instrukcije čitaju argumente iz rezervacionih stanica, prosleđujući emitovane operande i izvršavaju se. U ovoj implementaciji su registarski fajlovi rezervacionih stanica manji od centralnog registarskog file-a. Rezultati se upisuju u reorder buffer, rezervacione stanice ako su potrebni i u future file ako daju zadnji upis u arhitekturalni registar. Pri pojavi izuzetaka ili pogrešnog predviđanja grananja, što se otkriva kod commit-a, arhitekturalni file kopira se u future file i svi registri postaju ready u rename file-u. Alternativa je šema oporavka iz 8.4.4.

## *8.5. Odnos optimizacije u prevođenju i run time paralelizacije superskalarnih procesora*

Planiranje u vreme prevođenja omogućava dugoročno planiranje i optimizaciju, ali uz nedovoljno poznate zavisnosti po podacima, u nekim slučajevima. Kompleksnost hardvera superskalarnog procesora zahteva da se ograniči veličina instrukcijskog prozora, pa se time ne može uraditi dugoročno planiranje. Zato je za najbolje performanse neophodno raditi sve vidove optimizacije. Verovatno najinteresantniji deo optimizacije kod prevodioca je da, u situaciji kada postoji run time određivanje zavisnosti, u vreme prevođenja se možemo osloniti na statističke podatke o postojanju zavisnosti. Zato, kada nismo sigurni da zavisnosti postoje, možemo se osloniti na pretpostavku da zavisnost ne postoji i tako sprovoditi optimizaciju. Ako smo pogrešili u proceni zavisnosti, superskalarni procesor će samo da radi sporije, ali će raditi tačno uprkos pogrešnoj proceni postojanja zavisnosti u vreme prevođenja.

## *8.6. Današnji procesori*

Kako dominiraju superskalarni procesori, prvo će biti navedene ključne osobine procesora u pogledu paralelizma. Spekulativno izvršavanje sa out-of-order issue, execution i writeback podržavaju Pentium 3, 4 i M procesori, PowerPC G3, G4 i G5, MIPS 10000 i 12000, AMD K6, Athlon i Opteron. Dakle, svi najpoznatiji superskalarni procesori.

Broj instrukcija koje se izdaju na izvršavanje za većinu je 4 instrukcije po ciklusu – dakle kao u našem primeru. Starije verzije Pentium 4 procesora i Pentium 3 procesori i njihovi pandani kod AMD mogu da izdaju po 3 instrukcije po ciklusu. Međutim, već starije verzije Pentium 4 procesora imaju najveći instrukcijski prozor (126 instrukcija) i najveći broj registara koji se adresiraju pomoću tagova (128 registara). Novije verzije Pentium 4 procesora imaju i veće vrednosti pojedinih parametara, jer se povećavao broj tranzistora sa povećanjem gustine pakovanja na integrisanim kolima. Zanimljivo je da kompleksna logika superskalarnih mašina dovodi do značajnog povećanja minimalne dubine niza protočnih stepeni koje mora da prođe svaka instrukcija. U slučaju ranijih P4 procesora, ta vrednost već je bila 22 odnosno 24 stepena, zavisno od vrste operacije. Noviji P4 procesori imaju još veću minimalnu dubinu niza protočnih stepeni. Navodi se minimalna dubina, jer se sa ulaskom u instrukcijski prozor prelazi u fazu čekanja da argumenti postanu data ready, a to zavisi od zavisnosti po podacima.

Prediktori grananja superskalarnih procesora danas pogađaju sa verovatnoćom od preko 95%, ali se konstrukcije prediktora čuvaju kao poslovne tajne. S obzirom na ukupan broj protočnih stepeni i broj instrukcija koje se istovremeno izdaju u izvršavanje, prediktor grananja postaje ključna komponenta za performanse procesora.

Jedini šire rasprostranjeni procesori koji koriste VLIW, odnosno EPIC arhitekturu su Itanium procesori. Detaljnija analiza ovih procesora nije predmet ove monografije, ali svi principi paralelizacije u vreme prevođenja navedeni u prethodnim poglavljima ove knjige primenjeni su u prevodiocima za Itanium mašine.

## *8.7. Zaključak*

Modeli i principi paralelizacije prikazani u prethodnim poglavljima univerzalni su i nisu bili zavisni od implementacije. Iz njih su proizilazile potpuno drugačije implementacije u procesorima ili prevodiocima. Ti osnovni principi i modeli će biti osnova bar još nekoliko generacija procesora, bar kada se radi o instrukcijskom nivou paralelizma. Neki od modela, poput paralelizma DoAll petlji mogu se primeniti i na paralelizam na nivou procesa.