SADRŽAJ

1. PROJEKTOVANJE PROCESORA 4
1.1. Projektovanje ISP-a 4
1.2. Arhitektura, implementacija i realizacija 5
1.2.1. ISA 5
1.2.2. Dinamičko-statički interfejs 6
1.3. Performanse procesora - gvozdeni zakon 7
1.3.1. Optimizacija performansi procesora 7
1.4. Paralelno procesiranje na nivou - instrukcije 8
2. PROJEKTOVANJE PROTOČNOG PROCESORA 9
2.1. Balansiranje rada protočnih stepeni 9
2.1.1. Kvantizacija stepena 10
2.1.2. Hardverski zahtevi 12
2.2. Unificiranje tipova instrukcija 15
2.2.1. Klasifikacija tipova istrukcija 15
2.2.2. Spajanje zahteva za korišćenjem resursa 17
2.2.3. Implementacija protočne obrade 19
2.3. Minimiziranje protočnih zastoja 21
2.3.1. Programske zavisnosti i protočni hazardi 21
2.3.2. Identifikacija protočnih hazarda 23
2.3.3. Rešavanje protočnih hazarda 26
2.3.4. Smanjenje cene uvodjenjem puteva premošćavanja 28
2.3.5. Implementacija protočnog deblokiranja 30
2.4. Procesori sa dubljom protočnošću 37
3. ORGANIZACIJA SUPERSKALARNIH PROCESORA 40
3.1. Ograničenja skalarnih protočnih procesora 40
3.1.1. Neefikasna unifikacija u jedinstveni protočni sistem 41
3.1.2. Gubitak perfomansi zbog stroge protočnosti 41
3.2. Od skalarnih ka superskalarnim protočnim sistemima 42
3.2.1. Paralelna protočnost 43
3.2.2. Raznovrsne protočne obrade 46
3.2.3. Dinamička protočna obrada 47
3.3. Pregled principa rada superskalarnih protočnih mašina 52
3.3.1. Pribavljanje instrukcija 52
3.3.2. Dekodiranje instrukcija 54
3.3.3. Dispečovanje instrukcija 58
3.3.4. Izvršenje instrukcija 61
3.3.5. Kompletiranje i izvlačenje instrukcija 64
4. SUPERSKALARNE TEHNIKE 67
4.1. Tehnike za povećanje protoka instrukcija 67
4.1.1. Upravljanje tokom programa i kontrolne zavisnosti 67
4.1.2. Degradacija performansi zbog grananja 68
4.1.3. Tehnike za predikciju grananja 71
4.1.4. Oporavljanje nakon pogrešne predikcije kod grananja 75
4.1.5. Napredne tehnike za predikciju grananja 78
4.1.6. Druge tehnike za upravljanje tokom programa 82
4.2. Tehnike za registarski-protok-podataka 84
4.2.1. Ponovno korišćenje registara i lažnih *zavisnosti po podacima* 84

4.2.2. Tehnike za preimenovanje registara 85
4.2.3. Prave zavisnosti po podacima i ograničenja u protoku podataka 89
4.2.4. Klasični *Tomasulo* algoritam 91
4.2.5. Dinamičko *Execution* jezgro 98
4.2.6. Rezervacione stanice i bafer preuredjenja 101
4.2.7. Dinamički planer instrukcija 105
4.2.8. Druge tehnike za regulisanje toka podataka prema registrima 106
4.3. Tehnike za ubrzanje toka podataka prema memoriji 108
4.3.1. Instrukcije za pristup memoriji 108
4.3.2. Hijerarhijska organizacija memorije 111
4.3.3. Uredjenje memorijskih pristupa 121
4.3.4. Premošćavanje i prosledjivanje kod instrukcije *Load* 123
4.3.5. Druge tehnike za memorijski protok 128

1. PROJEKTOVANJE PROCESORA

Procesori koji su u stanju da izvršavaju unapred definisani skup instrukcija (*Instruction Set Processors - ISPs*) nazivaju se **mikroprocesori**. Funkcionalnost mikroprocesora karakteriše se skupom instrukcija koje je on u stanju da izvrši. Svi programi koji se izvršavaju na tom mikroprocesoru kodirani su pomoću tog skupa instrukcija. Unapred definisani skup instrukcija naziva se skup instrukcija te arhitekture (*Instruction Set Architecture-ISA*). *ISA* se koristi kao interfejs izmedju softvera i hardvera, ili izmedju programa i procesora. Sa aspekta metodologije projektovanja *ISA* se odnosi na *specifikaciju* dizajna, dok mikroprocesor ili ISP predstavlja *implementaciju* dizajna.

1.1. Projektovanje ISP-a

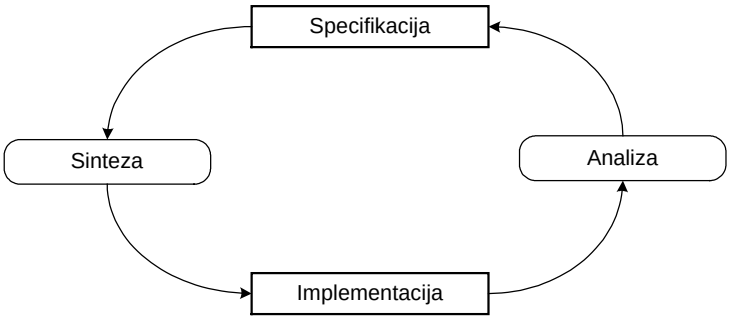
Svaki inženjerski dizajn počinje sa specifikacijom cilja koga treba ostvariti. **Specifikacija** se pre svega odnosi na opis ponašanja onoga što želimo da ostvarimo i predstavlja odgovor na pitanje "šta taj dizajn radi ".

Sa druge strane, **implementacija** se tiče strukturnog opisa rezultujućeg dizajna i daje odgovor na pitanje "kako je taj dizajn konstruisan".

Obično proces projektovanja čine dva fundamentalna zadatka: *sinteza* i *analiza*. Sinteza pokušava da pronadje implementaciju koja se zasniva na specifikaciji. Analiza ispituje implementaciju kako bi odredila da li i kako ona ispunjava specifikaciju.

Sinteza je kreativniji zadatak jer pretražuje moguća rešenja i pravi različite kompromise u optimizaciji dizajna kako bi se stiglo do najboljeg rešenja. Suština anlize je u odredjivanju korektnosti i efikasnosti dizajna, ona koristi *simulatore* da bi procenila performanse i validirala dizajn.

Jedan tipičan proces projektovanja koji, da bi se stiglo do konačnog cilja, zahteva višestruki prolaz kroz ciklus analiza-sinteza je prikazan na slici 1.1.



Slika 1.1 Inženjerski dizajn

Proces projektovanja mikroprocesora je veoma kompleksan i ne tako direktan kao na slici 1.1 Specifikacija dizajna mikroprocesora predstavlja *ISA* koja specificira skup instrukcija koje taj mikroprocesor mora da bude u stanju da izvrši. Implementacija je aktuelni hardverski dizajn koji se opisuje pomoću HDL-a. Proces projektovanja savremenih visoko-performansnih mikroprocesora čine sledeća dva koraka:

a) **mikroarhitekturni dizajn** – odnosi se na razvoj i definiciju ključnih tehnika, radi postizanja ciljnih performansi. Krajnji rezultat mikroarhitekturnog dizajna predstavlja opis organizacije mikroprocesora na visokom nivou. Opis obično koristi RTL (*Register Transfer Language*) da bi specificirao sve glavne module u organizaciji mašine, kao i interakciju izmedju ovih modula.

b) **logički dizajn** – RTL opis se detaljno razradjuje inkorporiranjem implementacionih detalja, kako bi se dobio HDL opis aktuelnog hardvera

1.2. Arhitektura, implementacija i realizacija

Kada se govori o arhitekturi računara definišu se tri osnovna i različita nivoa apstrakcije:

1) **arhitekturni** – specificira funkcionalno ponašanje procesora,

2) **implementacioni** – odnosi se na logičku strukturu ili organizaciju koja važi za tu arhitekturu, i

3) **realizacioni** – predstavlja fizičku strukturu u koju se ugradjuje ta implementacija.

Arhitektura se često naziva *ISA*. Ona speificira skup instrukcija koji karakterišu funkcionalno ponašanje *ISP*-a. Sav softver se mora preslikati ili kodirati u ovaj skup instrukacija kako bi mogao da se izvršava od strane procesora. Svaki program se kompajlira u sekvencu instrukcija koje pripadaju tom skupu. Tipični primeri arhitektura su IBM 360, DEC VAX, Motorola 68k, Power PC, IA 32, i dr.
Atributi koji prate arhitekturu su asemblerski jezik, format instrukcija, adresni način rada, i programski model.

Implementacija predstavlja specifični dizajn arhitekture koja se naziva mikroarhitektura. U toku života, jedna *ISA* arhitektura može da ima veći broj implementacija, pri čemu sve implementacije izvršavaju programe za tu *ISA*. Primeri implementacija poznatih arhitektura su IBM 360/91, VAX 11/780, Motorola 68040, PowerPC 604, Intel P6. Atributi koji su tipični za implementaciju su protočni dizajn, keš memorija, i prediktori granjanja. Implementacija se ostvaruje na nivou hardvera i nevidljiva je softveru.

Realizacija implementacije se odnosi na fizičko ugradjivanje dizajna. Kada se govori o mikroprocesorima tu se pre svega misli na to da li je to fizičko ugradjivanje sprovedeno na čipu ili multičipu. Za datu implementaciju postoji veći broj relizacija. Te realizacije se mogu razlikovati u odnosu na taktnu frekvenciju, kapacitet memorije, interfejs magistrale, pakovanje i dr. Atributi koji prate realizaciju su površina čipa, disipacija, hladjenje, pouzdanost, i dr.

1.2.1. ISA

ISA se koristi kao specifikacija za projektante procesora. Projektovanje mikroprocesora počinje sa *ISA*, a kao rezultat se dobija mikroarhitektura koja zadovoljava specifikacije. Svaka nova mikroarhitektura mora da se validira u odnodu na *ISA* kako bi se osigurali da li ona ispunjava funkcionalne zahteve specificirane od strane *ISA*. Ovo je od izuzetne važnosti ako želimo dokazati da će se postojeći softver korektno izvršavati na novoj mikroarhitekturi.

Do danas je razvijen, i koristi se, veliki broj *ISA*. One se razlikuju u odnosu na to kako se specificiraju operacije i operandi.

Obično *ISA* definiše skup instrukcija nazvan **asemblerske instrukcije**. Svaka instrukcija specificira operaciju i jedan ili veći broj operanada. Svaka *ISA* na jedinstven način definiše **asemblerski jezik**. Program na asemblerskom jeziku čini sekvenca asemblerskih instrukcija. Uglavnom *ISA* se medjusobno razlikuju po tome koliki se broj operanada specificira instrukcijom. Tako na primer, postoje 3-, 2-, 1-, i 0-adresne mašine.

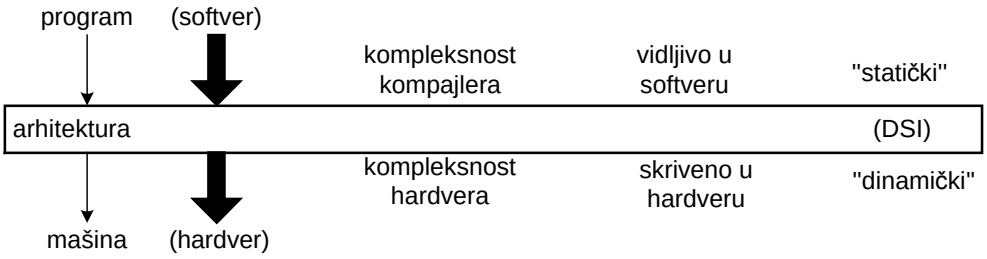
Za razvoj efikasnih kompajlera i operativnih sistema potreban je period od desetak godina. Ovo znači da je inercija *ISA* u odnosu na razvoj softvera dosta velika. U principu što je životni vek *ISA* veći to je softverska baza za tu *ISA* veća, pa je takvu *ISA* teže zameniti. Nasuprot sporom napretku na polju *ISA* inovacija, nove mikroarhitekture se razvijaju na svakih 3 do 5 godina.

1.2.2. Dinamičko-statički interfejs

Do sada smo ukazali na sledeće dve ključne uloge koje ima *ISA*:

1. ostvarivanje sprege izmedju hardvera i softvera što obezbedjuje nezavisni razvoj programa i mašina, i
2. specifikaciju dizajna mikroprocesora, sve implementacije moraju da ispune zahteve i podrže funkcionalnost specificiranu od strane *ISA*

No pored ove uloge postoji i treća kojom se definiše **dinamičko-statički interfejs** (*DSI*), tj. šta se od strane *ISA* obavlja statički u toku kompajliranja programa, a šta dinamički u toku izvršenja programa. Mesto i uloga *DSI*-a prikazana je na slici 1.2.



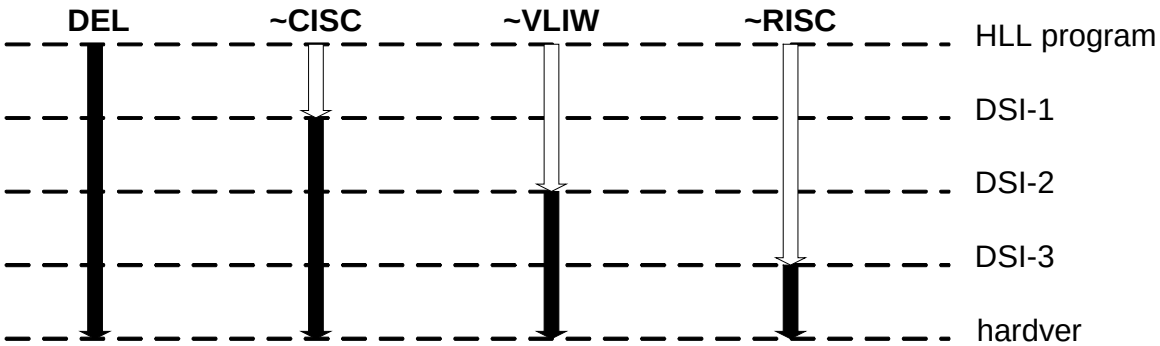
Slika 1.2 Dinamičko-statički interfejs

U dosadašnjem razvoju *ISA* dizajna, predložen je veći broj različitih razmeštaja *DSI*-a (vidi sliku 1.3). Tako na primer, *Mike Flynn* je predložio postavljanje *DSI*-a veoma visoko, pri čemu se sve aktivnosti obavljaju ispod *DSI*-a, tj. programi su napisani na HLL-u i mogu se izvršavati od strane mašine direktno (*directly executable language machine*).

CISC *ISA* smeštaju *DSI* na tradicionalnom asemblerskom jeziku, ili makrokôd, nivou.

Nasuprot tome RISC *ISA* spuštaju *DSI* i očekuju da se najveći broj optimizacija obavi od strane kompajlera.

Svrha trenda spuštanja *DSI*-a ka nižim nivoima je smanjenje kompleksnosti hardvera i realizacija mašina koje brže izvršavaju programe.



Slika 1.3 Konceptualni prikaz mogućih razmeštaja *DSI*-a kod *ISA* dizajna

1.3. Performanse procesora - gvozdeni zakon

Performanse procesora se definišu na osnovu **gvozdenog zakona** *(iron low)* koji glasi:

$$\frac{1}{performanse} = \frac{vreme}{program} = \frac{instrukcije}{program} * \frac{ciklusi}{instrukcije} * \frac{vreme}{ciklusi} \quad \text{.....(1.1)}$$

U principu performanse se mere u zavisnosti od toga koliko dugo je vremena potrebno da se neki program izvrši na toj mašini. Kraće vreme ukazuje na bolje performanse. Analizom jednačine (1.1) možemo da zaključimo sledeće:

a) Prvi član (*instrukcija/program*) ukazuje na ukupan broj dinamičkih instrukcija koji je potreban da se izvrši od strane programa, ovaj član se naziva **broj instrukcija** (*instruction count*).
b) Drugi član se odnosi na uprosečavanje, tj. koliko je mašinskih ciklusa potrebno za izvršenje svake instrukcije. Ovaj član se izražava u **ciklusima-po-instrukciji**, *CPI*, (C*ycles Per Instruction*).
c) Treći član ukazuje na vremensko trajanje svakog mašinskog ciklusa, tj. **ciklusa mašine** (*machine cycle*)

Performanse procesora se mogu povećati smanjenjem vrednosti svakog od tri člana u jednačini (1.1). No sva tri člana nisu nezavisna, naime, izmedju njih postoji složena interakcija. Smanjenje jednog člana može da dovede do povećanja ostala dva. Zbog ovoga, poboljšanje performansi predstavlja jedan izazov koji se sastoji u nalaženju dobrog kompromisa (balansa) izmedju svih faktora koji su od uticaja.

1.3.1. Optimizacija performansi procesora

Kao što smo već ukazali tehnike za optimizaciju performansi imaju za cilj da u jednačini (1.1) redukuju vrednost jednog ili većeg broja članova. Neke od tehnika mogu redukovati samo jedan član ostavljajući ostala dva nepromenjena. Tako na primer, kompajler obavlja optimizaciju putem eliminisanja redudantnih i beskorisnih instrukcija u objektnom kôdu, što znači da se može smanjiti broj instrukcija u programu (uticaj prvog člana u jednačini (1)), a da se pri tome ne promene *CPI* i vreme ciklusa (drugi i treći član). Jedan drugi primer bi bio sledeći. Ugradnjom bržih kola skraćuje se vreme propagacije signala kroz gejtove, a to znači da se potencijalno skraćuje trajanje mašinskog ciklusa, a da se pri tome ne promeni broj instrukcija i *CPI*. Ovakve tehnike za optimizaciju performansi su uvek poželjne i treba da se koriste samo kada cena koja treba da se plati zbog njihovog uvodjenja nije tako visoka.

Postoje i druge tehnike koje redukuju jedan od članova u jednačini (1), ali istovremeno povećavaju drugi ili ostala dva. Kod ovih tehnika, dobitak u performansama se postiže samo tada kada efekat redukcije jednog člana premašuje povećanje koje nastaje usled uticaja ostala dva. Analizirajmo sada poznate tehnike za redukciju članova, tj. optimizaciju performansi.

Postoji veći broj pristupa za redukciju broja instrukcija. Skup instrukcija može da sadrži složenije instrukcije koje obavljaju veći obim posla po instrukciji. Na ovaj način ukupan broj instrukcija potreban za izvršenje programa se smanjuje. Tako na primer, za izvršenje jednog programa na mašini tipa RISC *ISA* je potrebno dva ili veći broj puta dinamičkih instrukcija u odnosu na mašinu tipa CISC *ISA*. No smanjenje broja dinamičkih instrukcija često rezultira povećanju složenosti izvršne jedinice, a indirektno i povećanju vremena ciklusa.

Želja za smanjenjem *CPI*-a je inspirisala razvoj većeg broja arhitekturnih i mikroarhitekturnih tehnika. Jedna od ključnih motivacija kod RISC-ova odnosi se na smanjenje složenosti svake instrukcije sa ciljem da se redukuje broj mašinskih ciklusa potreban za procesiranje svake instrukcije.

Drugo rešenje za smanjenje *CPI*-a zasniva se na uvodjenju protočnosti. Treće rešenje se bazira na ugradnji keš memorije, čime se u značajnoj meri skraćuje prosečno vreme pristipa memoriji.

Ključna mikroarhitekturna tehnika za redukciju vremena ciklusa je protočnost. Protočnost efektivno deli zadatak procesiranja instrukcije na veći broj stepeni. U tom slučaju latencija u funkciji propagacije signala kroz protočni stepen odredjuje vreme trajanja mašinskog ciklusa.

1.4. Paralelno procesiranje na nivou - instrukcije

Paralelno procesiranje na nivou–instrukcije (*instruction-level parallel processing*) može se formalno definisati kao konkurentno procesiranje većeg broja instrukcija. Tradicionalno, sekvencijalni (CISC) procesori u datom trenutku izvršavaju po jednu instrukciju, i u proseku za procesiranje jedne instrukcije troše 10 mašinskih ciklusa, što znači da je $CPI = 10$.

Kod protočnih (RISC) procesora, i pored toga što je za kompletiranje izvršenja svake instrukcije potreban veći broj ciklusa, zahvaljujući preklapanju u izvršenju izmedju većeg broja instrukcija, *CPI* se redukuje na vrednost bliskoj jedinici.

Kod superskalarnih procesora koji pribavljaju i iniciraju izvršenje većeg broja instrukcija po mašinskom ciklusu $CPI < 1$. (Veoma često kod ovih procesora performanse se izražavaju preko **broja instrukcija** (*Instruction Count*, tj. kao $IPC = 1/CPI$).

Jednačinu (1) možemo zapisati i u sledećem obliku:

$$Performanse = \frac{1}{broj\ instrukcija} * \frac{instrucija}{ciklusi} * \frac{1}{vreme\ ciklusa} =$$

$$= \frac{IPC * frekvencija}{broj\ instrukcija} \quad \text{.....(1.2)}$$

Analizom jednačine (1.2) možemo zaključiti da se performanse mogu povećati povećanjem *IPC*-a, povećanjem frekvencije, ili smanjenjem broja instrukcija.

2. PROJEKTOVANJE PROTOČNOG PROCESORA

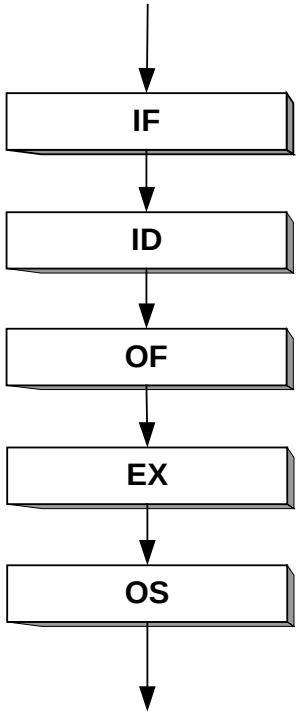
Tri primarna izazova koje treba ostvariti u toku projektovanja protočnog sistema su:

1. **uniformno izračunavanje** $\Rightarrow$ balansiran rad protočnih stepena
2. **identično izračunavanje** $\Rightarrow$ unificiranje tipova instrukcija
3. **nezavisna izračunavanja** $\Rightarrow$ minimiziranje zastoja u protočnoj obradi

2.1. Balansiranje rada protočnih stepeni

Jedan tipičan ciklus instrukcije se može funkcionalno podeliti na sledeće pet aktivnosti (izračunavanja):

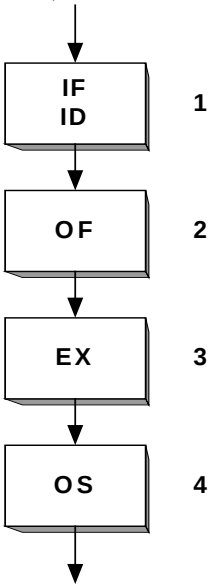
1. **pribavljanje instrukcije** (*instruction fetch*, **IF**)
2. **dekodiranje instrukcije** (*instruction decode*, **ID**)
3. **pribavljanje operan(a)da** (*operand(s) fetch*, **OF**)
4. **izvršenje instrukcije** (*instruction execution*, **EX**)
5. **smeštaj operanda** (*operand store*, **OS**)



Slika 2.1 Peto-stepeni GENERIC protočni sistem

2.1.1. Kvantizacija stepena

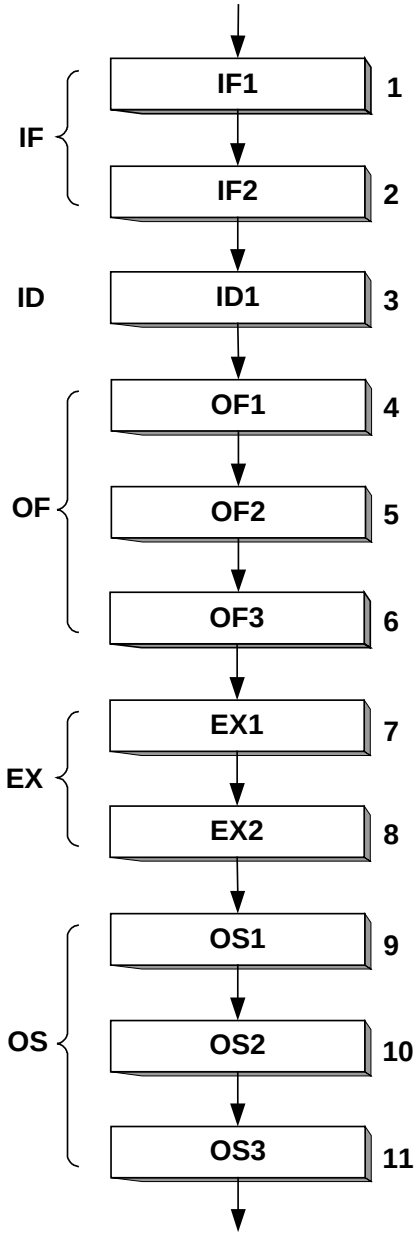
Sa ciljem da se postigne dobar balans u izračunavanju veći broj stepeni, čija je latencija kraća, može se grupisati u nova izračunavanja. Tako na primer, imajući u vidu da *ISA* koristi instrukcije fiksnog formata, jednostavne adresne načine rada, i ortogonalna polja u instrukcionom formatu, izračunavanja u stepenima IF i ID, koja su relativno jednostavnija u odnosu na ostala tri sa slike 2.1, mogu se grupisati u jedinstveno (vidi sliku 2.2.)



Slika 2.2 Četvorostepeni protočni sistem

Ovakav pristup rezultira mašinom sa grubo-zrnastim mašinskim ciklusom, a shodno tome i nižim stepenom protočnosti . Umesto da se kombinuju (spajaju) izračunavanja sa kraćim latencijama, veoma često, da bi se izbalansirao rad protočnih stepeni, usvaja se suprotan pristup. Naime, izračunavanje sa dugom latencijom se može podeliti na izračunavanja sa kraćim latencijama. Na ovaj način se dobija mašina sa sitno-zrnastim mašinskim ciklusom i visokim stepenom protočnosti.

Na slici 2.3 prikazan je protočno organizovan sistem koga čine 11 stepena.



Slika 2.3 11-stepeni protočni sistem

Zadatak

Kao što smo ukazali kvantizacija stepena se može ostvariti:
a) spajanjem većeg broja izračunavanja u jedinstveno (slika 2.2), ili
b) razbijanjem izračunavanja na veći broj manjih izračunavanja (slika 2.3)

Neka ukupna latencija 5-stepenog sistema sa slike 2.1 iznosi 280 ns, a rezultujuća vremena mašinskog ciklusa za 4-stepeni (slika 2.2) i 11-stepeni sistem (slika 2.3) iznose 80 ns i 30 ns, respektivno.

Za 4-stepeni i 11-stepeni protočni sistem odrediti:
a) ukupnu latenciju,
b) internu fragmentaciju,
c) propusnost.

Odgovor

a) Latencije 4- i 11-stepenih protočnih sistema iznose

$T_{4L} = 4 * 80\ ns = 320\ ns$
$T_{11L} = 11 * 30\ ns = 330\ ns$

b) razlika izmedju nove ukupne latentnosti i početno zadate ukupne latentnosti od $280\ ns$ predstavlja **interna fragmentacija**

$T_{4IF} = 320 - 280 = 40\ ns$ -za 4-stepeni sistem
$T_{11IF} = 330 - 280 = 50\ ns$ -za 11-stepeni sistem

c) Propusnost 4-stepenog sistema
$B_4 = \frac{280\ ns}{80\ ns} = 3{,}5$ puta u odnosu na neprotočni dizajn, dok je propusnost 11-stepenog sistema
$B_{11} = \frac{280\ ns}{30\ ns} = 9{,}3$ puta u odnosu na neprotočni dizajn

2.1.2. Hardverski zahtevi

Kod najvećeg broja realističnih dizajn rešenja cilj nije samo da se ostvare najbolje moguće performanse nego i da se postigne najbolji odnos performanse/cena. To znači da pored maksimiziranja propusnosti (povećanja performansi) protočnog sistema, moramo voditi računa i o hardverskim zahtevima (cena).
Zahtevi koji se odnose na realizaciju hardvera, a tiču se rada protočnog sistema, možemo svrstati u sledeće tri kategorije:
1. neophodna logika za upravljanje i manipulaciju podacima u svakom od stepena,
2. portovi *RF* polja koji su neophodni radi konkurentnog pristupa registrima od strane većeg broja stepeni,
3. memorijski portovi koji podržavaju konkurentni pristup memoriji od strane većeg broja stepeni.

Analizirajmo sada rad 4-stepenog protočnog sistema sa slike 2.1. Usvojimo da je to *Load /Store* arhitektura kod koje:

- jedna tipična instrukcija tipa registar-registar treba da čita dva registarska operanda u prvom stepenu, a smešta rezultat u registar u četvrtom stepenu
- *Load* instrukcija treba da čita iz memorije u drugom atepenu, dok *Store* instrukcija treba da upisuje u memoriju u četvrtom stepenu
- Prvi stepen treba u svakom ciklusu da čita (pribavlja) iz memorije po jednu instrukciju

Kombinovanjem zahteva od strane sva četiri stepena za pristup memoriji, i usvajući da se koristi unificirana memorija (jedinstvena memorija za instrukcije i podaci) zaključujemo da memorija u svakom mašinskom ciklusu mora da podrži dva pristupa radi čitanja i jedan radi upisa. Takodje u svakom mašinskom ciklusu postoji zahtev da se dvaput pristupa registarskom (*RF*) polju radi čitanja, a jedanput radi upisa.

Slična analiza koja se odnosi na hardverske zahteve može se sprovesti i za 11-stepeni protočni sistem sa slike 2.3.

- Da bi se usaglasio rad sa sporom memorijom izračunavanje u okviru IF stepena se deli i preslikava na dva protočna stepena,tj. na IF1 i IF2. Pri tome pribavljanje instrukcije se inicira u IF1, a završava u IF2. To znači da u toku svakog mašinskog ciklusa memorija za instrukcije mora biti u stnju da podrži dva konkurentna pristupa (jedan od strane IF1, a drugi od IF2).
- Na sličan način razbijanje stepena OF i OS na po tri stepena (OF1, OF2 i OF3, kao i OS1, OS2 i OS3) ukazuje da u datom trenutku u protočnom sistemu mogu da se nalaze do šest instrukcija koje pristupaju memoriji za podatke. To znači da memorija za podatke mora biti u stanju da podrži do šest nezavisnih konkurentnih pristupa, a da pri tome ne dodje do konflikta u svakom mašinskom ciklusu, tj. zahteva se ugradnja 6-to portne memorije za podatke. U principu multiportne memorije su izuzetno skupe za implementaciju. Nešto jevtinija rešenja su ona koja se baziraju na *interleaved* memoriji sa većim brojem banaka. Ove memorije simuliraju funkcionalnost pravih multiportnih memorija, ali ne garantuju uvek konkurentni pristup bez konflikta.

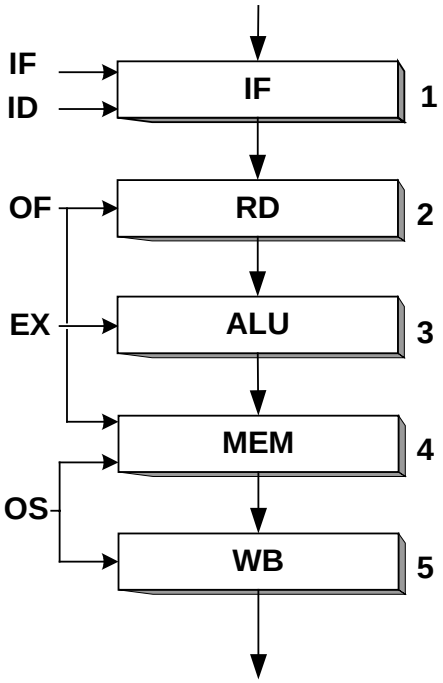
Na osnovu dosadašnje analize mogli bi da zaključimo sledeće:

- sa povećanjem dubine protočnosti iznos hardverskih resursa koji je neophodan da se podrži ovakva protočnost se takodje povećava.
- najznačajnije povećanje hardverskih resursa predstavljaju dodatni portovi za pristup *RF* polju i memorijskim jedinicama. Ovi portovi treba da podrže povećani iznos konkurentnih pristupa memorijskim jedinicama.
- da bi se uskladio rad sporog pristupa memoriji (duga latentnost) sa brzim radom protočnog stepena (kratka latentnost) aktivnosti koje se odnose na pristup memoriji treba uprotočiti. Na žalost, fizičko uprotočavanje memorijskih pristupa za više od dva mašinska ciklusa postaje veoma složeno, pa je veoma teško rešiti probleme koji se odnose na konkurentne pristupe bez konflikta.

Primer

U skladu sa prethodnom diskusijom sagledajmo hardversku složenost 5-stepenog protočno organizovanog mikroprocesora MIPS R2000 / R3000 prikazan na slici 2.4.

MIPS je *Load / Store* arhitektura, kod koje su izračunavanja IF i ID (sa slike 2.1) ujedinjena u jedinstveno. Stepen IF (slika 4), u svakom mašinskom ciklusu, izdaje po jedan zahtev za pristup memoriji za instrukcije (I-kešu). OF izračunavanje (sa slike 1) u konkretnom slučaju (slika 2.1) se obavlja u stepenima RD i MEM. *ALU* instrukcije manipulišu sa registarskim operandima. Pristup ovim operandima se vrši u stepenu RD, kada se i čitaju sadržaji oba dva registra. Kod *Load* instrukcije, za pribavljanje operanda potreban je jedan pristup memoriji (D-kešu), a ovaj pristup se izvodi u stepenu MEM (jedini stepen protočnog sistema koji može da pristupi D-kešu). OS izračunavanje (slika 2.1) u konkretnom slučaju (slika 2.1) se izvodi u stepenima MEM i WB. Instrukcija *Store* pristupa D-kešu u stepenu MEM. Instrukcije tipa *ALU* i *Load* upisuje rezultat u *RF* polju u stepenu WB (slika 2.1).



Slika 2.4 5-stepeni protočni sistem MIPS R2000 / R3000

Tekući trend koji se odnosi na projektovanje protočnih procesora usmeren je ka realizaciji procesora koji koriste duboku protočnost. Za prvu generaciju RISC protočnih procesora karakteristična je bila 4- ili 5-stepena obrada, dok je za današnje procesore svojstvena 10-stepena ili dublja obrada.

2.2. Unificiranje tipova instrukcija

Drugi aspekt protočnog idealizma polazi od činjenice da protočni sistem repetitivno obavlja jedno isto izračunavanje. Ali, u najvećem broju slučajeva ova pretpostavka ne važi. U suštini i pored toga što protočni sistem repetitivno izvršava instrukcije ipak se one medjusobno razlikuju. Naime različiti tipovi instrukcija imaju zahteve za korišćenjem različitih resursa, a takodje i ne zahtevaju istu sekvencu izračunavanja. Svaka instrukcija u toku obrade ne zahteva upotrebu svih protočnih stepeni, tako da neki od njih postaju pasivni. Ovaj tip neefikasnosti u radu naziva se **eksterna fragmentacija**. Ključni izazov projektantu je da minimizira eksternu fragmentaciju za sve tipove instrukcija.

2.2.1. Klasifikacija tipova istrukcija

Da bi obavio izračunavanja, računar mora biti u stanju da obavlja sledeća tri osnovna zadatka:

1. **aritmetičko-logičke operacije**,
2. **kopiranje (premeštanje) podataka**,
3. **sekvenciranje instrukcije**

Na koji način će se ova tri osnovna zadatka dodeliti različitim instrukcijama *ISA* predstavlja ključnu komponentu projektovanja skupa instrukcija. Primera radi kod CISC arhitektura postoje složene instrukcije tako da jedna instrukcija obavlja po dva ili sva tri zadatka. Sa druge strane kod RISC-ova svaka instrukcija izvršava samo jedan od tri osnovna zadatka.

U zavisnosti od prethodno pomenutih osnovnih zadataka, sve instrukcije se mogu klasifikovati u sledeće tipove:

1. ***ALU* instrukcije** – obavljaju sve aritmetičke i logičke operacije,
2. ***Load/Store* instrukcije** – premeštaju podatke izmedju registara i memorijskih lokacija,
3. ***Branch* instrukcije** – kontrolišu sekvenciranje instrukcija

ALU instrukcije obavljaju aritmetičke i logičke operacije, i to striktno nad registarskim operandima. Samo instrukcije *Load* i *Store* mogu pristupiti podacima u memoriji. Instrukcije tipa *Load*, *Store* i *Branch* koriste relativno jednostavne adresne načine rada. Tipično se podržava registarsko-indirektni način rada sa ofsetom. Standardno *Branch* instrukcije podržavaju PC-relativni adresni način rada. Kod sva tri tipa instrukcija pristup instrukcijama vrši se preko I-keša, a podacima preko D-keša.

Semantike *ALU*, *Load-Store* i *Branch* instrukcija, kojim se specificiraju sekvence izračunavanja definisane tim tipovima instrukcija, prikazanih na slikama 2.5, 2.6 i 2.7, respektivno.

Izvorno izračunavanje	**Tip ALU instrukcije**	
	Integer instrukcija	FP instrukcija
IF	• pribavljanje instrukcije (pristup I-kešu)	• pribavljanje instrukcije (pristup I-kešu)
ID	• dekodirane instrukcije	• dekodiranje instrukcije
OF	• pristup registru *RF* polja	• pristup registru FP polja
EX	• obavlja se ALU operacija	• obavlja se FP operacije
OS	• upis rezultata u registar *RF* polja	• upis rezultata u registar FP polja

Slika 2.5 Specifikacija instrukcije tipa *ALU*

Izvorno izračunavanje	**Instrukcije tipa *Load / Store***	
	Instrukcija *Load*	Instrukcija *Store*
IF	• pribavljanje instrukcije (pristup I-kešu)	• pribavljanje instrukcije (pristup I-kešu)
ID	• dekodirane instrukcije • pristup registru *RF* polja (bazna adresa)	• dekodiranje instrukcije
OF	• generisanje efektivne adrese (baza + ofset) • pristup radi čitanja memorijske lokacije (pristup D-kešu)	• pristup registru *RF* polja (registarski operand i bazna adresa)
EX	--	-- • generisanje efektivne adrese (baza + ofset)
OS	• upis rezultata u registar *RF* polja	• pristup radi upisa u memorijsku lokaciju (pristup D-kešu)

Slika 2.6 Specifikacija instrukcija tipa *Load / Store*

Izvorno izračunavanje	**Instrukcije tipa *Branch***	
	Jump (bezuslovna) instrukcija	Uslovna *Branch* instrukcija
IF	• pribavljanje instrukcije (pristup I-kešu)	• pribavljanje instrukcije (pristup I-kešu)
ID	• dekodirane instrukcije • pristup registru *RF* polja (bazna adresa)	• dekodiranje instrukcije • pristup registru *RF* polja (bazna adresa)
OF	• generisanje efektivne adrese (baza + ofset)	• generisanje efektivne adrese (baza + ofset)
EX	--	• procena uslova grananja
OS	• ažuriranje vrednosti PC-a sa ciljnom adresom	• ako je uslov istinit, ažurira se vrednost PC-a sa ciljnom adresom

Slika 2.7 Specifikacija instrukcije tipa *Branch*

Analizirajući specifikacije za sva tri tipa instrukcija (slika 2.5, 2.6 i 2.7) vidimo da su inicijalna (izvorna) izračunavanja identična (aktivnosti u okviru stepena IF i ID), dok su ostale specifičnosti sledeće:

- *ALU* instrukcije ne pristupaju memoriji za podatke, tj. ne generišu memorijske adrese .
- *Load/Store* i *Branch* instrukcije imaju iste zahteve u pogledu izračunavanja efektivne adrese.
- *Load/Store* instrukcije mora da pristupaju memoriji za podatke, dok instrkcija *Branch* mora da generiše adresu ciljne instrukcije grananja.
- Kod uslovnih instrukcija grananja (*Conditionl Branch*), pored zahteva za generisanjem efektivne adrese, mora da se obavi procena uslova grananja. Ova procena se obavlja proverom stanja statusnog bita koji je postavljen od strane prethodne instrukcije.

Konačno bi mogli da zaključimo sledeće: I pored toga što sva tri tipa instrukcija obavljaju identične aktivnosti u stepenima IF i ID, izračunavanja u ostalim stepenima (OF, EX i OS) se razlikuju, a to dovodi do razlike u zahtevima za korišćenjem resursa.

2.2.2. Spajanje zahteva za korišćenjem resursa

Izazov da se unificiraju različiti tipovi instrukcija podrazumeva efikasno spajanje različitih zahteva za korišćenjem tih resursa u jedinstveni instrukcioni protočni sistem koji može da se prilagodi svim tipovima instrukcija. Cilj je da se minimizira ukupan broj resursa, a maksimizira korišćenje tih resursa.

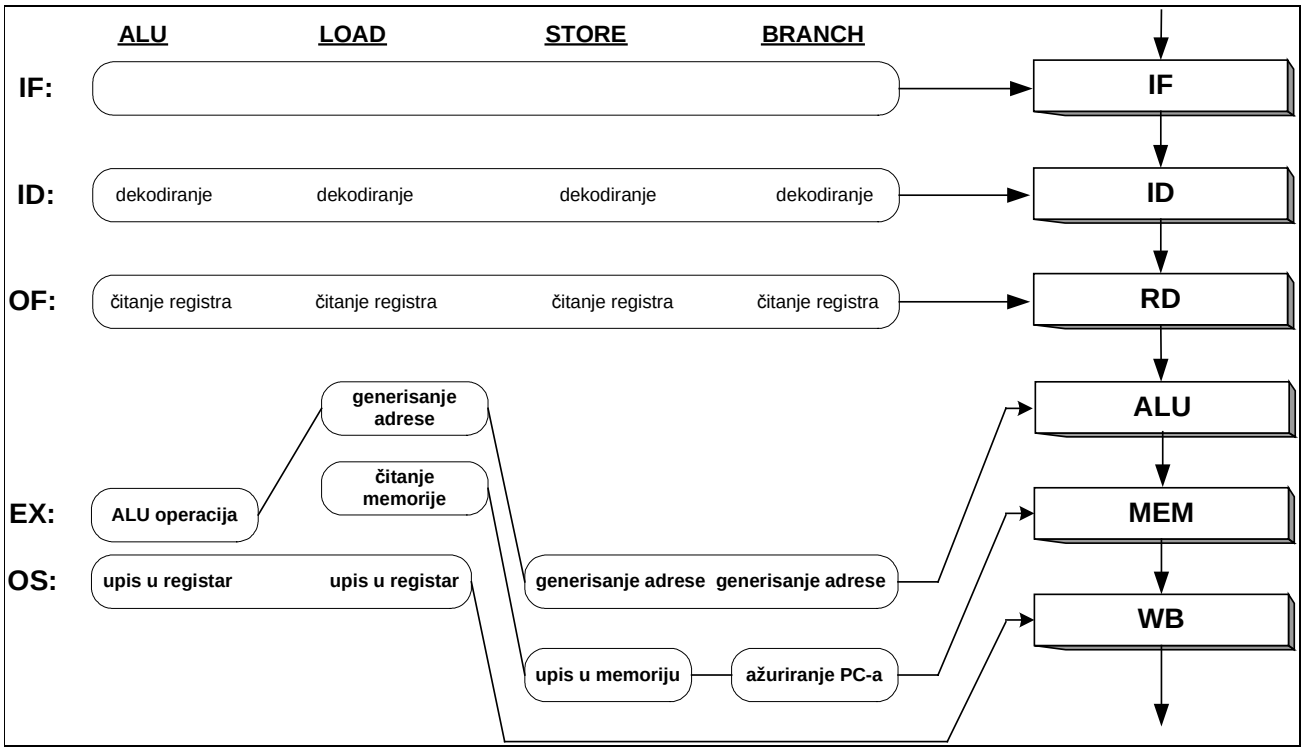
Procedura za unificiranje različitih tipova instrukcija neformalno se sastoji od sledećih koraka:

1. Analiziranje sekvence izračunavanja za svaki tip instrukcije, i identifikacija odgovarajućih zahteva za korišćenjem resursa,
2. Pronalaženje sve što je zajedničko izmedju tipova instrukcija, i spajanje zajedničkih izračunavanja sa ciljem da dele isti protočni sistem
3. Ako postoji fleksibilnost, bez da se naruši semantika instrukcija, treba pomeriti redosled izračunavanja kako bi se olakšalo dalje spajanje.

Shodno slikama 2.5, 2.6 i 2.7 na slici 2.8 za jedan 6-stepeni prtotočni sistem prikazane su sumarne specifikacije za instrukcije tipa *ALU*, *Load*, *Store* i *Branch.*

Analizirajući sliku 2.5 mogli bi da zaključimo sledeće:

a) sva četiri tipa instrukcija obvljaju ista izračunavanja u stepenima IF i ID tako da ovi stepeni mogu biti zajednički.
b) sve četiri instrukcije čitaju iz *RF* polja (odnosi se na OF stepen sa slike 1). *ALU* instrukcije pristupaju *RF* polju radi pribavljanja po dva registarska operanda. *Load* i *Branch* instrukcije pristupaju registru da bi dobavili baznu adresu. *Store* instrukcija pristupa jednom registru *RF* polja da bi dobavila registarski operand, a drugom registru radi bazne adrese. U sva četiri slučajeva čita se sadržaj jednog ili dva registra. Shodno ovim sagledavanjima sva ova slična izračunavanja se mogu spojiti u treći protočni stepen koga ćemo nazvati RD (vidi sliku 2.5). *RF* polje mora biti u stanje da svakog mašinskog ciklusa podrži dve nezavisne konkurentne operacije čitanja.



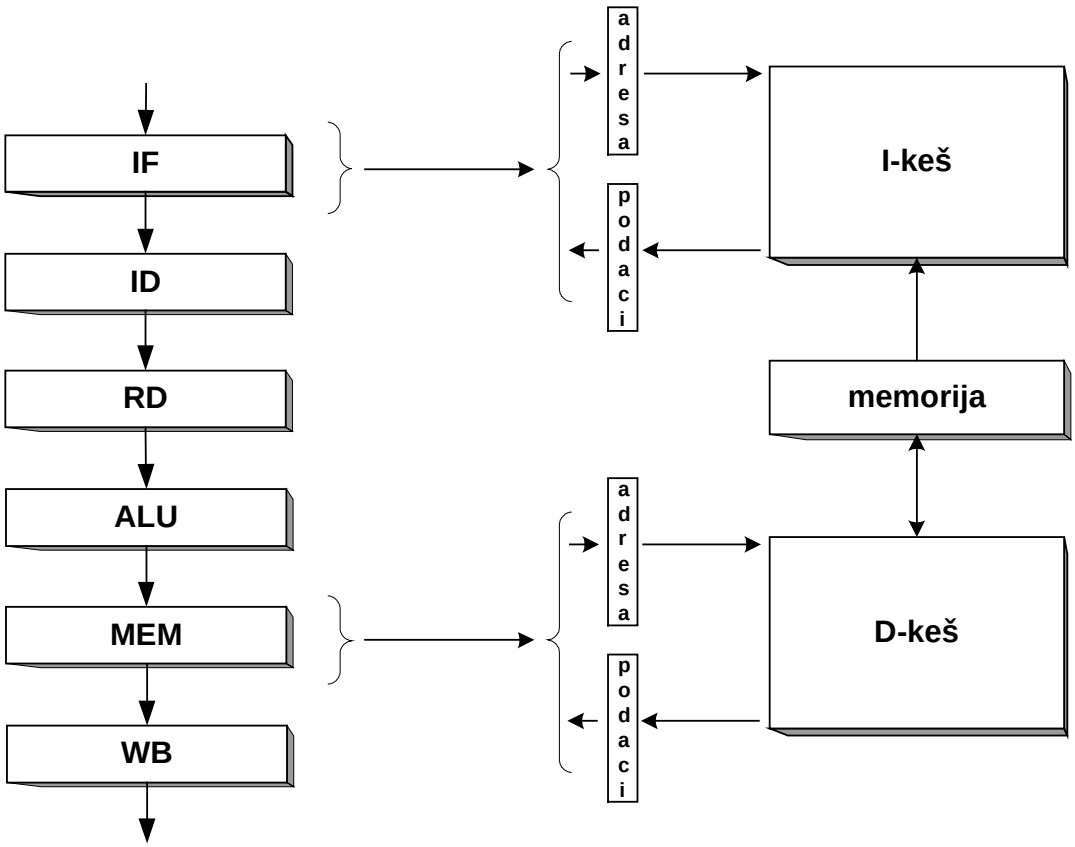
Slika 2.8 Unificiranje instrukcije tipa *ALU, Load, Store i Branch* kod 6-stepenog protočnog sistema skraćeno nazvan TYP

c) *ALU* instrukcije zahtevaju ALU funkcionalnu jedinicu koja obavlja neophodne aritmetičke i logičke operacije. Za potrebe instrukcija *Load*, *Store* i *Branch* funkcionalna jedinica ALU se koristi za generisanje efektivne adrese radi pristupa memoriji. To znači da se sva ova izračunavanja mogu spojiti u četvrti protočni stepen nazvan ALU (vidi sliku 2.8).
d) Instrukcije *Load* i *Store* treba da pristupaju memoriji za podatke, što znači da se jedan protočni stepen, nazvan MEM (vidi sliku 2.8), mora dodeliti za ove aktivnosti
e) Instrukcije tipa *ALU* i *Load* moraju, u toku zadnjeg koraka izračunavanja, da upišu rezultat u odredišni registar *RF* polja. *Load* instrukcija puni odredišni registar podatkom koji se pribavlja iz memorije za podatke u stepenu MEM. Sa druge strane instrukcija tipa *ALU* je već generisala rezultat u stepenu ALU (vidi sliku 2.8), ali radi unificiranja sa instrukcijom *Load* upis rezultata u odredišni registar *RF* polja se vrši u stepenu WB. To znači da se rezultat instrukcije ALU samo prenosi kroz stepen MEM, tj. kasni za jedan mašinski ciklus, i tek nakon toga u stepenu WB upisuje u *RF* polje. Naime za instrukciju tipa *ALU* stepen MEM unosi jedan pasivan mašinski ciklus.
f) Kod uslovnih instrukcija grananja, pre ažuriranja stanja PC-a neophodno je odrediti uslov grananja. S obzirom da se ALU funkcionalna jedinica koristi za izračunavaje efektivne adrese ona se ne može koristiti i za procenu uslova grananja. Za slučaj da se uslov grananja odnosi samo na proveru da li je vrednost nekog registra nula, pozitivna ili negativna, tada je dovoljno ugraditi samo jedan jednostavni komparator. Kao gradivni blok, komparator se može ugraditi što je moguće pre u protočnom lancu obrade, a to je stepen ALU, koji sledi nakon čitanja sadržaja referentnog registra koje se obavlja u stepenu RD. To znači da najkraći stepen u kome se sdržaj PC-a može ažurirati na vrednost ciljne adrese grananja (izračunata je ciljna adresa grananja i procenjen uslov grananja) predstavlja stepen MEM (vidi sliku 2.8).

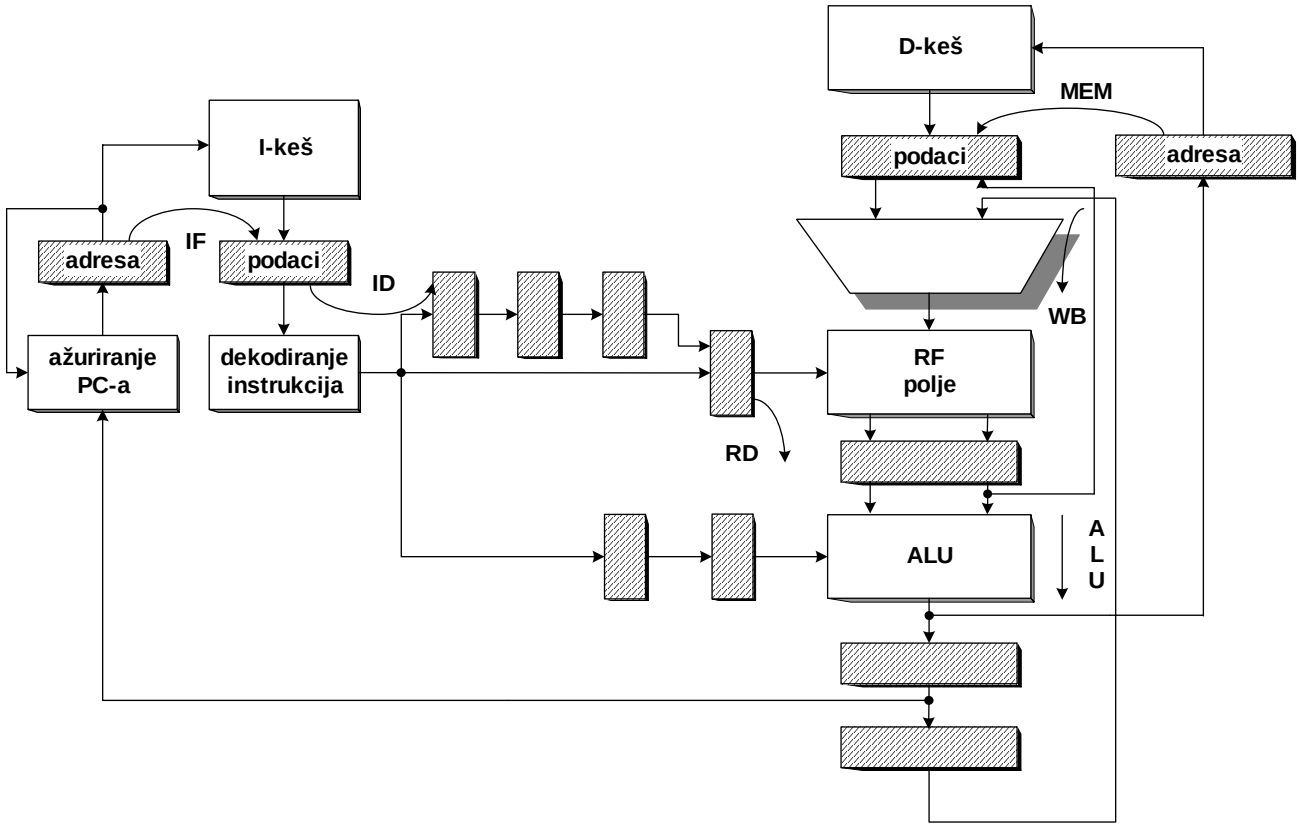
Analizirajući sliku 2.8 zaključujemo da je 6-stepeni protočni sistem nazvan TYP veoma efikasan, jedino instrukcija *Load* aktivno koristi sva šest stepena, dok ostale tri instrukcije aktivno koriste po pet od šest stepeni. U datom trenutku sistem TYP može istovremeno da izvršava do šest različitih instrukcija, pri čemu se svaka instrukcija procesira od strane jednog od protočnih stepeni u svakom mašinskom ciklusu *RF* polje mora da podrži do dve operacije čitanja (od strane instrukcije u RD stepenu) i jednu operaciju upis (od strane instrukcije koja se nalazi u stepenu WB). I-keš, u svakom mašinskom ciklusu, mora da podrži jednu operaciju čitanja. Osim kod grananja, IF stepen mora kontinualno da inkrementira PC i pribavlja narednu sekvencijalnu instrukciju iz I-keša. D-keš, u svakom mašinskom ciklusu, mora da podrži jednu operaciju čitanja memorije ili upis u memoriju. Samo stepen MEM može da pristupa D-kešu što znači da samo jedna instrukcija u protočnom sistemu može da pristupa memoriji za podatke.

2.2.3. Implementacija protočne obrade

Fizička organizacija 6-stepenog protočnog sistema TYP prikazana je na slici 2.9. Kao što se vidi sa slike 2.9 postoje dva jedno-portna keša, D-keš i I-keš. IF stepen pristupa I-kešu, a MEM stepen D-kešu (za detalje vidi sliku 2.7). U svakom mašinskom ciklusu I-keš podržava pribavljanje po jedne instrukcije. Kod stepena MEM, instrukcija *Load*(*Store*) čita (upisuje) iz (u) D-keš. Usvojeno je da su latencije D-keša i I-keša po jedan mašinski ciklus.

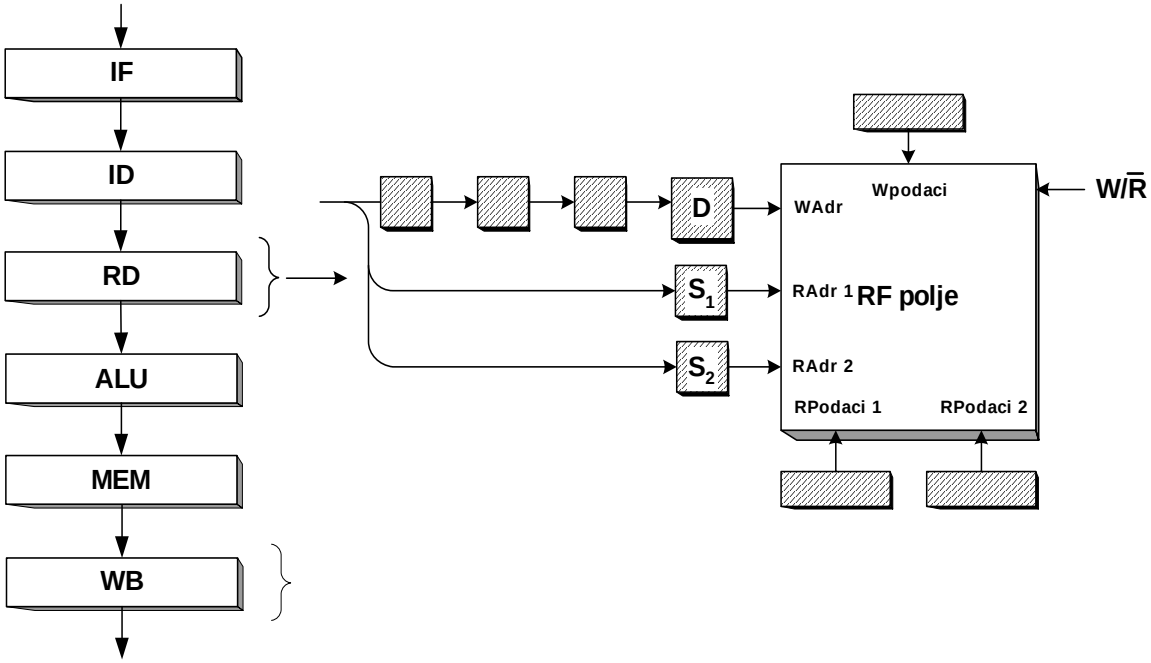


Slika 2.9 Interfejs prema memorijskom sistemu kod 6-stepenog protočnog sistema TYP



Slika 2.10 Fizička organizacija protočnog sistema TYP

Interfejs ka multi-portnom *RF* polju prikazan je na slici 2.10. Samo stepeni RD i WB pristupaju *RF* polju. U svakom mašinskom ciklusu *RF* polje mora potencijalno da podrži do dva čitanja registara od strane stepena RD, i jedan upis u registar od strane stepena WB. To znači da multi-portno *RF* polje ima dva izlazna porta za podatke i jedan ulazni port, takodje za podatke.



Slika 2.11 Interfejs ka multi-portnom *RF* polju kod 6 – stepenog protočnog sistema TYP

Kao što se vidi sa slike 2.11 postoje dodatna tri baferska stepena (osenčeni blokovi) povezani na registarskom adresnom portu za upis. Baferski stepeni obezbedjuju da adresa odredišnog registra, u kome se vrši upis, pristigne na ulaz porta, u istom trenutku, kao i podatak koga treba upisati (izlaz stepena WB koji se preko baferskog stepena (osenčeni blok) dovodi na ulaz *W_podaci RF* polja).

2.3. Minimiziranje protočnih zastoja

Treći aspekt idealizma u protočnoj obradi bazira se na činjenici da su sva izračunavanja koja se obavljaju od strane protočnog sistema nezavisna. Kod k-stepene protočne obrade postoji k različitih izračunavanja koja se istovremeno obavljaju. No izmedju instrukcija postoje zavisnosti koje uzrokuju zastoje u radu protočnog sistema. U daljem tekstu identifikovaćemo tipove zastoja i ukazaćemo na načine njihovog rešavanja.

2.3.1. Programske zavisnosti i protočni hazardi

Na ISA abstraktnom nivou, program se specificira kao sekvenca asemblersko jezičkih instrukcija. Jedna tipična instrukcija se specificira kao funkcija:

$$i : T \leftarrow S_1 \; op \; S_2$$

gde je:

$D(i) = \{S_1 , S_2\}$ - **domen instrukcije** i ; $R(i) = \{T\}$ predstavlja **opseg instrukcije** i ; a preslikavanje iz domena u opseg je definisano operacijom op.

Neka su date dve instrukcije i i j , pri čemu u leksičkom redosledu obeju instrukcija, j sledi i . Izmedju instrukcija i i j kažemo da postoji **zavisnost-po-podacima** (*data dependencies*), koju označavamo sa $i \; \delta \; j$, ako važi jedan od sledeća tri uslova:

$$R(i) \cap D(j) \neq 0 \qquad (1)$$

$$R(j) \cap D(i) \neq 0 \qquad (2)$$

$$R(i) \cap R(j) \neq 0 \qquad (3)$$

Prvi uslov ukazuje da instrukcija *j* zahteva operand koji se nalazi u opsegu instrukcija *i*. Ovo se naziva *read-after-write* (*RAW*) ili **prava** (*true*) **zavisnost po podacima**, i označava se sa $i\,\delta_d\,j$. Implikacija prave *zavisnosti po podacima* je takva da instrukcija *j* ne može da počne sa izvršenjem sve dok instrukcija *i* ne završi.

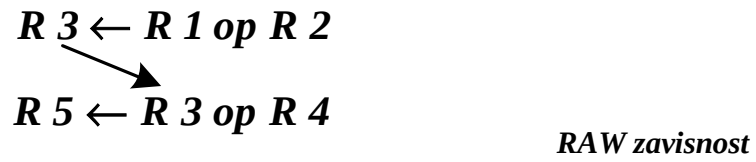
Drugi uslov ukazuje da se operand koji koristi instrukcija *i* nalazi u opsegu instrukcija *j*, ili da instrukcija *j* modifikuje promenljivu koja predstavlja operand instrukcije *i*. Ova zavisnost se naziva *write-after-read* (*WAR*) ili **anti zavisnost po podacima**, i označava se sa $i\,\delta_a\,j$. Egzistencija anti zavisnosti iziskuje da instrukcija *j* ne završi pre instrukcije *i*, inače će instrukcija *i* pribaviti pogrešan operand.

Treći uslov ukazuje da obe instrukcije *i* i *j* dele u svom opsegu zajedničku promenljivu, što znači da obe modifikuju tu promenljivu. Ova se naziva *write-after-write* (*WAW*) ili **izlazna zavisnost po podacima**, a označava se kao $i\,\delta_o\,j$. Postojanje izlazne zavisnosti iziskuje da prvo završi instrukcija *i*, a tek nakon toga instrukcija *j*, inače će instrukcije koje slede nakon instrukcije *j* koristiti pogrešan operand.

Treba naglasiti da postoji i zavisnost tipa *read-after-read*, što znači da obe instrukcije *i* i *j* pristupaju istom operandu radi čitanja. S obzirom da operacija čitanja nije destruktivnog tipa, (za razliku od operacije upis, operacija čitanja ne menja sadržaj operanda), za ovaj tip zavisnosti kažemo da nije opasan. Naime nije bitan relativni redosled čitanja jednog operanda od strane instrukcija *i* ili *j*, tj. da li se operand prvo čita od strane instrukcije *i*, a zatim od *j*, i obratno.

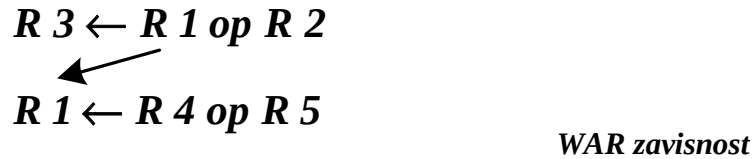
Na slici 2.12 prikazane su RAW, WAR i WAW registarske *zavisnosti po podacima*

- prava zavisnost



RAW zavisnost

- anti zavisnost



WAR zavisnost

- izlazna zavisnost

WAW zavisnost

Slika 2.12 RAW, WAR i WAW registarske zavisnosti

Imajući u vidu da u asemblerskom jeziku domeni i opsezi instrukcija mogu biti promenljive koje se nalaze u registrima ili memorijskim lokacijama, zavisnosti koje mogu da se jave mogu biti *memorijske zavisnosti* ili *registarske zavisnosti*.

Pored *zavisnosti po podacima* izmedju dve instrukcije mogu da postoje i **kontrolne zavisnosti**, ili tzv. *upravljačke zavisnosti*. Neka su date dve instrukcije *i* i *j*, pri čemu instrukcija *j* sledi nakon instrukcije *i*. Tada za instrukciju *j* kažemo da je upravljački zavisna od instrukcije *i*, što se označava sa $i\ \delta_c\ j$.Uslovne instrukcije grananja uzrokuju neodredjenost u sekvenciranju instrukcija. Naime instrukcije koje slede nakon uslovnog grananja mogu biti zavisne po upravljanju od instrukcije *Branch*.

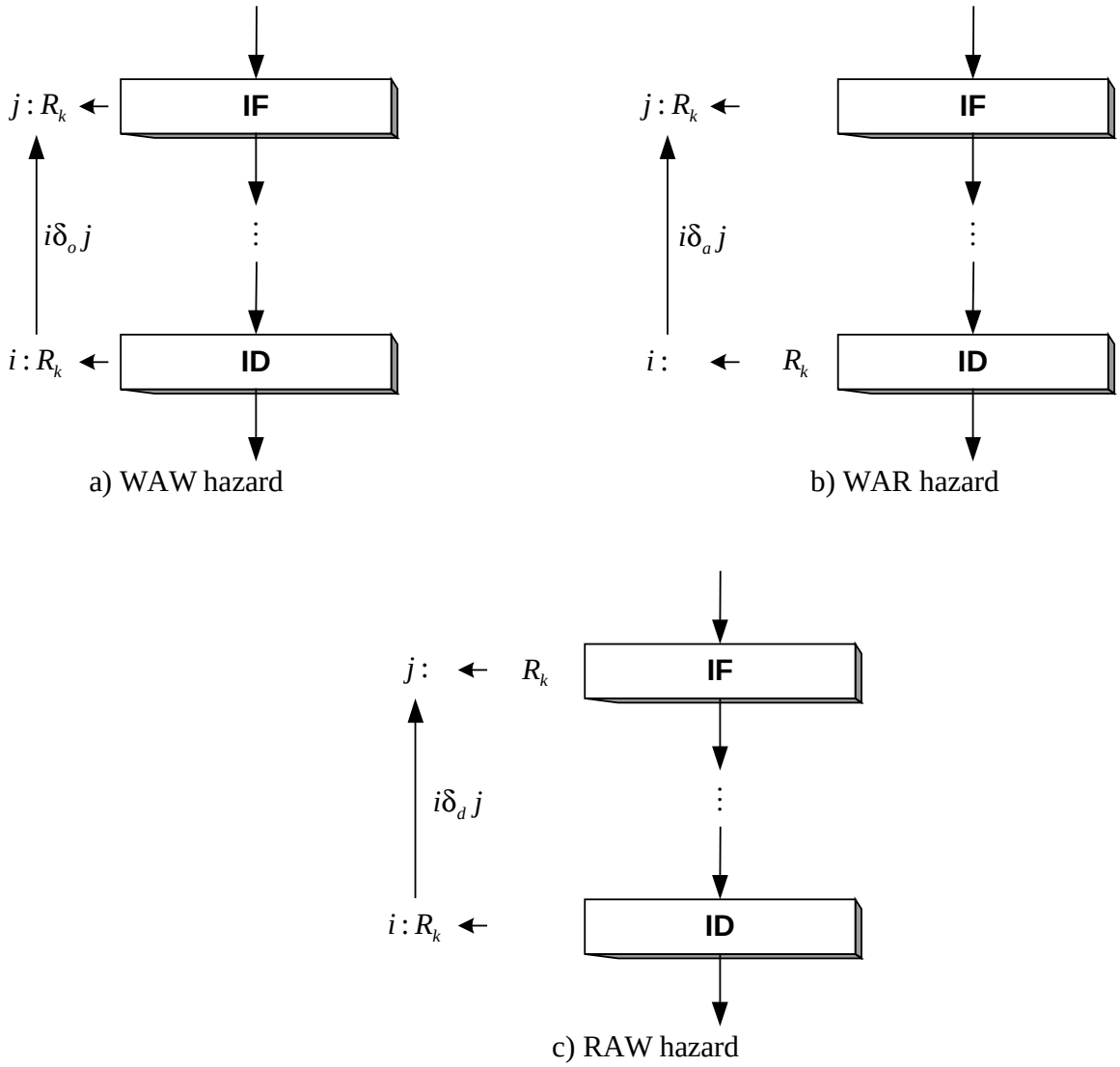
Program na asemblerskom jeziku čini sekvenca instrukcija. Korektnost semantike ovog programa zavisna je od sekvencijalnog izvršenja instrukcija. Sekvencijalni listing instrukcija ukazuje na sekvencijalno prethodjenje izmedju susednih instrukcija, tj. na redosled u izvršenju instrukcija. Ako u programskom listingu iza instrukcije *i* sledi instrukcija *i+1*, tada se implicitno podrazumeva da prvo treba da se izvrši instrukcija *i* , a nakon toga da sledi izvršenje instrukcije *i+1*.

Ako se ostvari ovakvo sekvencijalno izvršenje, tada za semantičku korektnost programa kažemo da je očuvana (zagarantovana). Ili nešto preciznije, pošto ciklus instrukcija čini veći broj izračunavanja (aktivnosti), implicitno se podrazumeva da će se sve aktivnosti instrukcije *i* obaviti pre nego što bilo koje izračunavanje instrukcije *i+1* može da počne. Ovaj način rada nazivamo **totalno sekvencijalno izvšavanje**, tj. sva izračunavanja predvidjena sekvencom instrukcija se izvode sekvencijalno.

Kod *k*-stepenog protočnog sistema postoji preklapanje u procesiranju izmedju *k* instrukcija. Naime, odmah nakon što je instrukcija *i* završila svoje prvo izračunavanje i počne sa drugim, instrukcija *i+1* počinje svoje prvo izračunavanje. U datom trenutku, kod *k-stepenog* protočnog sistema istovremeno se obavljaju *k* izračunavanja. To znači da režim rada totalno sekvencijalno izršenje ne važi. Dok totalno sekvencijalno izvršenje predstavlja dovoljan uslov da se očuva semantička korektnost programa, on ne predstavlja i potreban uslov (zahtev) za očuvanje semantičke korektnosti. Totalno sekvencijalno izvršenje uslovljeno sekvencijalnim listingom instrukcija predstavlja dodatnu specifikaciju programa. Suštinski zahtev koga treba ispuniti je da se semantika programa ne naruši, a to se može ostvariti ako se ne naruše zavisnosti izmedju instrukcija. Drugim rečima, ako izmedju instrukcija *i* i *j* postoji zavisnost, pri čemu u programskom listingu instrukcija *j* sledi nakon instrukcije *i*, tada operacija čitanje/upis zajedničke (deljive) promenljive za instrukciju *i* i *j* mora da se ostvari kako je to definisano u izvornom (originalnom) sekvencijalnom redosledu. Kod protočnih procesora, ako se o ovome ne vodi računa, postoji potencijalna opasnost da se naruše programske zavisnosti. Potencijalna narušavanja programskih zavisnosti se nazivaju **protočni hazardi**. Svi protočni hazardi moraju biti detektovani i razrešeni, ako se želi ostvariti korektno izvršenje programa.

2.3.2. Identifikacija protočnih hazarda

Protočni hazard predstavlja potencijalni prekršaj zavisnosti u programu. Da bi se u protočnom sistemu javio hazard treba da postoje najmanje dva stepena koja sadrže dve instrukcije koje istovremeno pristupaju zajedničkoj (istoj) promenljivoj. Na slici 2.13 ilustrovani su neophodni uslovi koji u protočno organizovanom sistemu dovode do pojave WAW, WAR i RAW hazarda. Uzrok pojave hazarda mogu biti memorijske i registarske *zavisnosti po podacima*. Na slici 2.13 ilustrovane su samo registarske zavisnosti. Napomenimo da i memorijske *zavisnosti po podacima* mogu biti tipa RAW, WAR i WAW.



Slika 2.13 Neophodni uslovi da se u protočnom sistemu jave WAW, WAR i RAW hazardi

Analizirajmo sada pod kojim uslovima kod 6-stepenog protočnog sistema TYP dolazi do pojave memorijskih i registarskih *zavisnosti po podacima*, kao i kontrolnih zavisnosti.

Memorijske *zavisnosti po podacima* se javljaju ako postoji zajednička promenljiva koja se čuva u memoriji, a pristupa joj se (bilo radi čitanja ili upisa) od strane dve instrukcije. Za datu *Load/Store* arhitekturu, memorijska zavisnost po podacima može da postoji samo izmedju instrukcija *Load/Store*. Da bi odredili da li protočni hazardi mogu da se jave zbog memorijske *zavisnosti po podacima* neophodno je da se ispita procesiranje instrukcija *Load/Store* od strane protočnog sistema. Ako usvojimo da je keš podeljen na I-keš i D-keš, tada samo stepen MEM može da pristupa D-kešu. Imajući u vidu da se svi pristupi memorijskim lokacijama od strane *Load/Store* instrukcija obavljaju od strane stepena MEM, tada realno postoji samo jedan stepen u protočno organizovanom sistemu koji može da upisuje i čita u/iz D-keša.

U saglasnosti sa prezentacijom na slici 2.13 do hazarda usled memorijske *zavisnosti po podacima* kod procesora TYP ne može da dodje. U suštini, svi pristupi D-kešu (memoriji za podatke) se obavljaju sekvencijalno, tako da je i procesiranje *Load/Store* instrukcija u potpunosti sekvencijalno. Na osnovu prethodne diskusije zaključujemo da kod TYP protočno organizovanog sistema ne postoje protočni hazardi zbog memorijske *zavisnosti po podacima*.

Razmotrićemo sada egzistenciju registarske zavisnosti. Da bi odredili protočne hazarde koji se mogu javiti zbog registarske *zavisnosti po podacima*, moramo prvo da identifikujemo sve protočne stepene koji pristupaju registarskom polju (*RF* polju). Kod procesora TYP, sve operacije čitanja registra obavljaju se u stepenu RD, a sve operacije upis u registar egzistiraju u stepenu WB.

Izlazna zavisnost (WAW), označena kao $i\ \delta_o\ j$ ukazuje da instrukcija *i* kao i naredna instrukcija *j* dele isti odredišni registar. Izlazna zavisnost postoji zbog toga što instrukcija *i* mora prvo da upiše rezultat izračunavanja u odredišni registar, a nakon toga instrukcija *j* mora da obavi upis u isti registar. Kod protočnog sistema TYP, samo stepen WB može da obavi upis u *RF* poplje . Shodno prethodnoj diskusiji, svi upisi u registar se obavljaju u sekvencijalnom redosledu od strane stepenaWB. Saglasno neophodnom uslovu sa slike 2.13 a), kod sistema TYP, hazardi usled izlazne zavisnosti ne mogu da se jave.

Anti-zavisnost (WAR), koja se označava kao $i\ \delta_a\ j$, ukazuje da instrukcija *i* čita iz registra koji za narednu instrukciju *j* predstavlja odredišni registar. Da ne bi došlo do hazarda moramo obezbediti uslove da instrukcija *i* čita registar pre nego što instrukcija *j* upisuje u taj registar. Jedini način da antizavisnost uzrokuje protočni hazard je kada naredna instrukcija *j* obavi upis u registar pre nego što instrukcija *i* pročita sadržaj tog registra. Ovakav scenario kod procesora TYP nije moguć jer sve operacije čitanja registra se obavljaju u stepenu RD, koji se u protočnoj organizaciji nalazi ispred stepena WB (WB je jedini stepen koji može da vrši upis u registar). Ova diskusija ukazuje da neophodni uslov sa slike 2.13 b) ne postoji kod procesora TYP. Shodno prethodnom, protočni hazardi zbog anti-zavisnosti ne mogu da se jave kod procesora TYP.

Jedini tip registarske *zavisnosti po podacima* koji kod procesora TYP može da dovede do pojave protočnih hazarda predstavlja prava zavisnost po podacima. Neophodni uslov, prikazan na slici 2.13 c), egzistira kod procesora TYP jer protočni stepen RD, koji obavlja čitanje registra, u protočnom lancu obrade, je pozicioniran ispred stepena WB, koji vrši upis u registar. Prava zavisnost po podacima, koja se označava kao $i\ \delta_d\ j$, podrazumeva da instrukcija *i* upisuje u registar dok instrukcija *j* koja sledi čita iz istog registra. Ako instrukcija *j* neposredno sledi instrukciju *i*, tada kada *j* dodje do stepena RD, instrukcija *i* će biti u stepenu ALU.

Zbog ovoga , instrukcija *i* ne može pročitati registarski operand koji je rezultat instrukcije *i* sve dok instrukcija *i* ne prodje kroz stepen WB. Da bi otklonili ovu zavisnost po podacima , instrukciji *j* mora da se zabrani ulaz u stepen RD sve dok instrukcija *i* ne završi procesiranje u stepenu WB. Na osnovu prethodne diskusije zaključujemo da se hazardi zbog prave zavisnosti, kod procesora TYP, mogu javiti iz razloga što naredna instrukcija može da stigne do protočnog stepena gde se vrši čitanje registra, pre nego što prva instrukcija u protočno organizovanom sistemu ne kompletira sa upisom u taj registar.

Konačno razmatraćemo kontrolne zavisnosti. Ovaj tip zavisnosti javlja se zbog uslovnih instrukcija grananja. Ishod uslovne instrukcije grananja odredjuje da li narednu instrukciju koju treba pribaviti predstavlja sledeću sekvencijalnu (po redosledu) instrukciju, ili je to ciljna instrukcija grananja. U suštini mogu da postoje dve kandidat instrukcije koje slede nakon instrukcije uslovnog grananja. Kod protočno organizovanog sistema, pod normalnim uslovima rada, IF stepen koristi sadržaj PC-a da bi pribavio narednu instrukciju, a zatim inkremetira PC da bi pokazao na narednu sekvencijalnu instrukciju, sve sa ciljem da se protočni lanac celo vreme održava popunjen. Ovaj zadatak se ponavlja u toku svakog mašinskog ciklusa. No kada se pribavi uslovna instrukcija grananja, može da dodje do potencijalnog narušavanja sekvencijalnoig toka. Ako do grananja ne dodje, tada već pribavljene instrukcije od strane stepena IF su korektne (nalaze se na pravom putu sekvenciranja).

Ali za slučaj kada do grananja dolazi, tada već pribavljene instrukcije od strane stepena IF nisu korektne (ne nalaze se na putu sekvenciranja). U suštini problem neodredjenosti se ne može uspešno rešiti sve dok se ne odredi ishod koji ukazuje na uslov grananja.

Kontrolna zavisnost se može posmatrati kao da predstavlja odredjeni oblik RAW zavisnosti, iz razloga što manipuliše sa sadržajem PC-a. Naime, uslovna instrukcija grananja upisuje u PC, dok se

pribavljanje naredne instrukcije sprovodi čitanjem sadržaja PC-a. Kada do grananja dodje, uslovna instrukcija grananja ažurira sadržaj PC-a na adresu ciljne instrukcije. Inače PC se ažurira na adresu naredne sekvencijalne instrukcije. Kod procesora TYP, ažuriranje PC-a na adresu ciljne instrukcije se obavlja u stepenu MEM, dok stepen IF koristi sadržaj PC-a da bi pribavio narednu instrukciju. Shodno prethodnom, IF stepen čita PC, dok stepen MEM, koji u protočnom lancu sledi kasnije, upisuje u PC. Redosled stepena IF i MEM u protočnom lancu, u saglasnosti sa slikom 2.13 c), zadovljava neophodan uslov da se pojave RAW hazardi, zbog manipulacije sa sadržajem registra PC. Na osnovu svega izloženog zaključujemo da kod protočnog sistema TYP kontrolna zavisnost postoji, i da se ova može posmatrati kao odredjeni oblik RAW hazarda koji podrazumeva manipulisanje sa sadržajem registra PC.

2.3.3. Rešavanje protočnih hazarda

Kako smo već konstatovali kod protočnog sistema TYP jedini protočni hazardi tipa zavisnost-po-podacima koji se mogu javiti su RAW hazardi.Pored toga mogu se javiti i hazardi zbog kontrolne zavisnosti. Svi ovi hazardi uzrokuju da vodeća instrukcija *i* upisuje u registar (ili PC), a instrukcija *j* koja sledi za njom čita taj registar. Da bi se uspešno rešili probleme koji prate hazarde, moraju se preuzeti odgovarajuće akcije. Pri ovome, efekti preuzetih akcija ne smeju da naruše odgovarajuće zavisnosti.

Sa tačke gledišta svakog RAW hazarda moramo obezbediti uslove da se operacija čitanja javi nakon operacije upis u **hazardni registar**.

Da bi uspešno rešili RAW hazard narednoj instrukciji *j* mora da se zabrani da udje u protočni sistem u kome se čita sadržaj hazardnog registra od strane instrukcije *j*, sve dok vodeća instrukcija *i* ne prodje protočni stepen u kome se u hazardni registar vrši upis od strane instrukcije *i*. Ovo se ostvaruje zaustavljanjem rada ranijih stepena protočne obrade, tj. svih stepena koji prethode stepenu u kome se vrši čitanje registra. Time se instrukciji *j* brani da udje u stepen u kome se obavlja čitanje kritičnog (hazardnog) registra. Broj mašinskih ciklusa za koji se koči procesiranje instrukcije *j*, u najgorem slučaju, jednako je rastojanju izmedju oba kritična stepena u protočnom sistemu, tj. stepena koji obavljaju čitanje i upis u hazardni registar, respektivno. Kada se govori o procesoru TYP, ako je vodeća instrukcija *i* tipa *ALU* ili *Load*, tada se upis u kritični registar vrši u stepenu WB, a čitanje kritičnog registra se obavlja u stepenu RD. Razlika izmedju ova dva stepena je tri, što znači da bi , u najgorem slučaju, zastoj trajao tri mašinska ciklusa (vidi sliku 2.14). Najgori slučaj se javlja kada instrukcija *j* neposredno sledi instrukciju *i*, što znači da je $j = i+1$. U ovom slučaju, na ulaz u stepen RD instrukcija *j* se mora zaustaviti u trajanju od tri mašinska ciklusa, a dozvoliće joj se da počne sa procesiranjem u stepenu RD kada instrukcija *i* napusti stepen WB. Ako naredna instrukcija *j* ne sledi odmah iza instrukcije *i* , tj. ako izmedju instrukcije *i* i *j* postoje druge instrukcije, tada cena koja mora da se plati zbog zastoja je manja od tri ciklusa, pod uslovom da instrukicije izmedju *i* i *j* ne zavise od instrukcije *i*. Stvarna cena, izražena u broju ciklusa zastoja, koja mora da se plati , jednaka je 3-*s*, gde je *s* broj instrukcija izmedju *i* i *j*. Tako na primer, ako je s = 3, tada cena koja treba da se plati je nula, što znači da *j* ulazi u stepen RD onog trenutka kada *i* napušta stepen WB, pa shodno tome, nije potrebno ubaciti zastoj da bi se zadovoljila cena RAW zavisnosti.

vodeća instrukcija tip (*i*)	**ALU**	**Load**	**Branch**
prateća instrukcija tipa (*j*)	**ALU, L/S,Br**	**ALU, L/S,Br**	**ALU, L/S,Br**
hazardni registar	Int. register (Ri)	Int. register (Ri)	PC
WRITE u registar u stepenu (*i*)	WB (stepen 6)	WB (stepen 6)	MEM (stepen 5)
READ registra u stepenu (*j*)	RD (stepen 3)	RD (stepen 3)	IF (stepen 1)
RAW rastojanje ili cena koja se mora platiti	3 ciklusa	3 ciklusa	4 ciklusa

Slika 2.14 Cena koja se u najgorem slučaju plaća zbog RAW hazarda kod protočnog procesora TYP

Kod upravljačkih hazarda, vodeća instrukcija *i* je instrukcija tipa *Branch*, koja ažurira PC u stepenu MEM. Pribavljanje prateće instrukcije *j* zahteva čitanje PC-a u stepenu IF. Razlika izmedju stepena MEM i IF je četiri, što znači da, u najgorem slučaju, cena koja mora da se plati iznosi četiri ciklusa. Kada se izvršava instrukcija uslovnog grananja, sva prethodna pribavljanja instrukcija se moraju stopirati zaustavljanjem rada stepena IF, sve dok se procesiranje uslovne instrukcije grananja ne kompletira u stepenu MEM, nakon čega se sadržaj PC-a ažurira na ciljnu adresu grananja. Ovo zahteva da se procesiranje u stepenu IF zaustavi za četiri ciklusa. Daljom analizom dolazimo do zaključka da je ovaj zastoj opravdan samo kada se uslovno grananje obavi. Kada do grananja ne dodje, tada stepen IF treba da produži sa pribavljanjem instrukcija koje u programskom redosledu slede kao naredne. Ova osobina se može ugraditi u protočni dizajn, tako da nakon instrukcije uslovnog grananja, dalje pribavljanje instrukcije se ne zaustavlja. To znači da protočni sistem usvaja da do grananja ne dolazi. U slučaju kada *do-grananja-dodje*, PC se ažurira na ciljnu adresu grananja u stepenu MEM, a zatim sve instrukcije koje se nalaze u protočnim stepenima koji prethode (kakvi su ALU, RD, ID i IF) se poništavaju, a naredna instrukcija se pribavlja sa ciljne adrese grananja. Kod ovakvog rešenja, cena od četiri ciklusa zastoja se plaća samo kada, kod uslovne instrukcije grananja, *do-grananja-dodje*, a u slučaju kada *do-grananja-ne-dodje* tada ne dolazi ni do pojave umetanja zastoja.

Slično RAW hazardima koji se javljaju zbog *zavisnosti–po-podacima*, i cena zastoja od četiri ciklusa koja se ploča kod kontrolnih hazarda može da se sagleda kao najgori slučaj RAW hazarda. No ako izmedju instrukcije *i* i instrukcije *j* mogu da se ubace instrukcije koje nisu kontrolno zavisne od instrukcije *i*, tada stvarni broj ciklusa zastoja koji se mora platiti može se smanjiti za broj umetnutih instrukcija. Ovaj koncept se naziva zakašnjena grananja (*delayed branches*). U suštini ciklusi zastoja se popunjavaju korisnim instrukcijama koje moraju da se izvrše nezavisno od ishoda grananja, tj. da li do grananja dolazi ili ne dolazi. Cena koja mora da se plati zbog zastoja iznosi 4-*s*, gde je *s* broj kontrolno-nezavisnih instrukcija koje se mogu umetnuti izmedju instrukcija *i* i *j*. Zakašnjeno grananje, ili popunavanje ciklusa kod grananja, unosi dodatne teškoće u implementaciji ranije pomenute tehnike, koja se bazira na činjenici da do grananja neće doći, koja dozvoljava stepenu IF da pribavi sve instrukcije koje u sekvencijalnom programskom redosledu slede nakon instrukcije grananja.

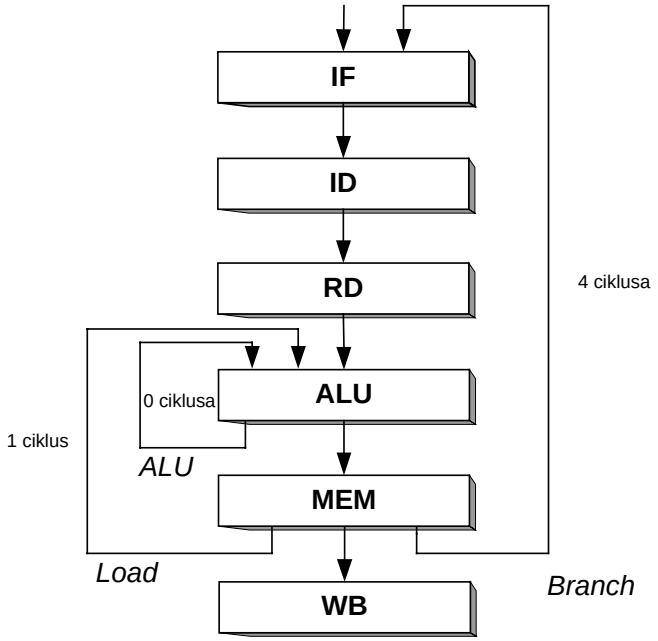
Razlog je taj da se moraju obezbediti mehanizmi koji će biti u stanju da prave razliku izmedju umetnutih instrukcija i instrukcija koje pripadaju normalnom sekvencijalnom redosledu. U slučaju kada *do-grananja-dodje* umetnute instrukcije ne treba izbacivati, ali treba poništiti (eliminisati efekat) onih instrukcija koje se nalaze na normalnom sekvencijalnom putu, iz razloga što one ne bi trebalo da se izvše.

2.3.4. Smanjenje cene uvodjenjem puteva premošćavanja

U dosadašnjoj analizi došli smo do zaključka da jedini mehanizam koji može da izadje na kraj sa hazardima predstavlja zaustavljanje napredovanja prateće instrukcije kroz protočni sistem. Zaustavljanjem se garantuje da će se operacije upis i čitanje u hazardni registar ostvariti u normalni sekvencijalni redosled. No često kod protočno organizovanih sistema za uspešno rešavanje problema hazarda koristi se jedna daleko agresivnija tehnika nazvana **premošćavanje puteva** (*forwarding paths*).

Na slici 2.15 prikazano je procesiranje vodeće instrukcije *i* za slučaj kada je *i* instrukcija tipa *ALU* ili *Load*. Ako je vodeća instrukcija *i* instrukcija tipa *ALU*, tada rezultat koji je potreban instrukciji *j*, a generiše od strane stepena ALU, postaje dostupan kada instrukcija *i* kompletira svoje izvršenje u stepenu ALU. Drugim rečima, operand koji je potreban instrukciji *j* u suštini je dostupan na izlazu ALU stepena u trenutku kada instrukcija *i* napusti stepen ALU, tako da *j* ne treba da čeka dva ciklusa da bi *i* napustila stepen WB. Ako se izlaz ALU stepena može učiniti dostupan ulazu ALU stepena preko fizičkih puteva za prosledjivanje (premošćavanje), odmah nakon što je on izračunat, tada će pratećoj instrukciji *j* biti dozvoljeno da udje u stepen ALU, onog trenutka kada vodeća instrukcija *i* napusti stepen ALU. Kod ovakve situacije, instrukcija *j* ne treba u stepenu RD da pristupa *RF* polju, kako bi dobavila (pročitala) zavisni operand. Umesto toga ona dobavlja zavisni operand sa izlaza ALU stepena.

Ugradnjom puta za premošćavanje zajedno sa odgovarajućom pratećom upravljačkom logikom, u najgorem slučaju, smanjujemo cenu, u broju zastoja, koja mora da se plati. Naime, cena sada iznosi nula ciklusa, za slučaj da je vodeća instrukcija *ALU* tipa. Šta više, i kada je prateća instrukcija instrukcija *i+1* nema potrebe za uvodjenjem zastoja, jer instrukcija *i+1* može da udje u stepen ALU onog trenutka kada instrukcija *i* napusti stepen ALU, što odgovara situaciji normalne protočne obrade.



Slika 2.15 Ugradnja puteva za premošćavanje kako bi se kod procesora TYP smanjila cena zbog *ALU* i *Load* instrukcije

Za slučaj kada je vodeća instrukcija tipa *Load*, a ne *ALU*, izvodi se slično premošćavanje. Naime kod instrukcije *Load* sadržaj memorijske lokacije koji se puni u registar postaje dostupan na izlazu stepena MEM, tj., kada instrukcija *Load* kompletira svoje izvršenje u ovom stepenu. Ponovo, put za premošćavanje se može ugraditi sa izlaza stepena MEM ka ulazu u stepen ALU, sa ciljem da podrži zahtev koji potiče od prateće instrukcije. To znači da prateća instrukcija može da udje u stepen ALU onog trenutka kada vodeća instrukcija kompletira svoje izvršenje u stepenu MEM. U ovoj situaciji, zbog vodeće instrukcije *Load*, cena zastoja od tri ciklusa (najgori slučaj) se redukuje na jedan ciklus. U najgorem slučaju, zavisna instrukcija može biti instrukcija *i+1*, tj. *j=i+1*. Tada u normalnoj procesnoj obradi, kada se instrukcija *i* nadje u stepenu ALU tada instrukcija *i+1* biće u stepenu RD. Kada instrukcija *i* advansira ka stepenu MEM, instrukcija *i+1* mora i dalje da se zadrži u stepenu RD zaustavljajući dalje napredovanje i onih instrukcija koje su već ušle u stepenima ID i IF. No u narednom ciklusu, kada instrukcija *i* napusti stepen MEM, zahvaljujući putu premošćavanja sa izlaza stepena MEM na ulaz stepena ALU, instrukciji *i+1* biće dozvoljeno da udje u stepen ALU. U suštini, procesiranje instrukcija *i+1* se zaustavlja za jedan ciklus u stepenu RD.

Cena koja se plaća zbog RAW hazarda kod TYP procesora sa izvedenim premošćavanjem za različite tipove instrukcija prikazana je na slici 2.16.

vodeća instrukcija tipa: (*i*)	**ALU**	**Load**	**Branch**
prateća instrukcija tipa:(*j*)	**ALU, L/S,Br**	**ALU, L/S,Br**	**ALU, L/S,Br**
hazardni registar	Int. register (Ri)	Int. register (Ri)	PC
READ registra u stepenu (*j*)	RD (stepen 3)	RD (stepen 3)	IF (stepen 1)
premošćavanje sa izlaza	ALU, MEM, WB	MEM, WB	MEM
premošćavanje ka ulazu	ALU	ALU	IF
cena koja se plaća u ciklusima zastoja	0 ciklusa	1 ciklus	4 ciklusa

Slika 2.16 Cena koja se u najgorem slučaju plaća zbog RAW hazarda kod procesora TYP kada se koriste putevi za premošćavanje

Cena koja se plaća zbog hazarda kada je ALU instrukcija videća naziva se **ALU cena** (*ALU penalty*). Na sličan način cena koja se plaća kada je *Load* instrukcija vodeća naziva se **Load cena** (*Load penalty*). Odredište puta premošćavanja (za *ALU* i *Load* instrukcije) predstavlja ulaz ALU. Put premošćavanja sa tačke (mesta) kada je rezultat najranije dostupan ka krajnjoj tački gde je taj rezultat zavisnoj instrukciji potreban niziva se **kritični-put-premošćavanja** (*critical forwarding path*), i on predstavlja najbolje rešenje sa tačke gledišta redukcije, koje se zbog postojanja hazarda plaća za taj tip instrukcije.

Pored kritičnog-puta-premošćavanja, neophodno je uvesti i dodatne puteve za premošćavanje. Tako na primer, neophodno je uvesti puteve premošćavanja sa izlaza stepena MEM i WB do ulaza u stepen ALU. Ova dva dodatna puta za premošćavanje su potrebna jer zavisna instrukcija *j* može potencijalno biti instrukcija *i+2* ili instrukcija *i+3*. Ako je *j=i+2* tada kada je instrukcija *j* spremna da udje u stepen ALU, instrukcija *i* biće na izlazu stepena MEM. Shodno tome, rezultat instrukcije *i*, koji još nije upisan u odredišni registar i potreban je instrukciji *j*, a dostupan je na izlazu MEM može se proslediti ka ulazu ALU stepena, čime se obezbedjuje uslov da instrukcija *j*, u narednom ciklusu, udje u stepen ALU. Na sličan način, ako je *j=i+3*, rezultat instrukcije *i* se može proslediti sa izlaza stepena

WB na ulaz stepena ALU. To odgovara situaciji kada je instrukcija *i* kompletirala upis u odredišni registar (stepen WB), a instrukcija *j* je prošla kroz stepen RD, i spremna je da udje u stepen ALU.

U slučaju kada je *j=i+4*, RAW zavisnost je lako zadovoljiti jer instrukcija *j* obavlja normalno čitanje sadržaja odredišnog registra *RF* polja, tj. nema potrebe za uvodjenjem puteva premošćavanja (u tom trenutku instrukcija *j* ulazi u stepen RD, a instrukcija *i* je već završila procesiranje u stepenu WB).

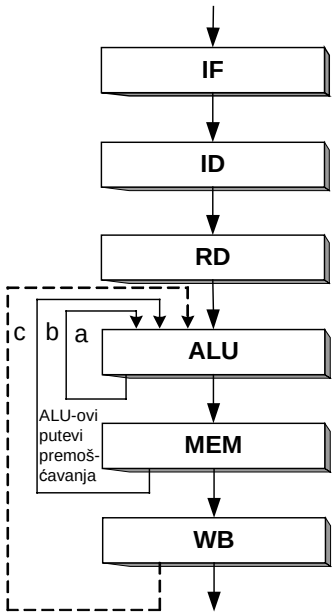
Kada je vodeća insrukcija tipa *Load*, najranije mesto gde je rezultat instrukcije *i* dostupan predstavlja izlaz stepena MEM, a krajnje mesto gde je taj rezultat potreban predstavlja ulaz ALU stepena. Shodno tome *kritični-put-premošćavanja* kod instrukcije *Load* predstavlja put od izlaza stepena MEM do ulaza u stepen ALU. Ovim premošćavanjem cena koja treba da se plati je jedan ciklus. Ponovo, još jedanput premošćavanje je potrebno sa izlaza stepena WB na ulaz u stepen ALU, što odgovara situaciji kada je prateća instrukcija *j* spremna da udje u stepen ALU, a vodeća instrukcija *i* izlazi iz stepena WB.

Analizirajući sliku 2.16 uočavamo da se ne koristi put premošćavanja kada treba da se plati cena zastoja zbog instrukcije *Branch*. Ako je vodeća instrukcija *i* tipa *Branch*, tada za raspoloživi način adresiranja, najranije mesto gde je taj rezultat dostupan predstavlja izlaz stepena MEM. Kod instrukcije *Branch*, odredjivanje ciljne adrese grananja kao i generisanje uslova grananja se vrši u stepenu ALU. Provera uslova grananja kao i punjenje ciljne adrese grananja u PC vrši se u stepenu MEM. To znači da je nakon stepena MEM moguće koristiti vrednost PC-a sa ciljem da se pribavi instrukcija sa ciljne adrese grananja. Sa druge strane, PC mora biti dostupan na početku stepena IF kako bi se obezbedilo pribavljanje naredne instrukcije. Na osnovu ovog zaključujemo da krajnje mesto gde je taj rezultat potreban predstavlja početak stepena IF. Kao rezultat kritični put premošćavanja, je onaj koji odgovara ceni koja mora da se plati za ažuriranje vrednosti PC-a na vrednost ciljne adrese grananja u stepenu MEM i početak pribavljanja ciljne adrese grananja u narednom ciklusu za slučaj da do grananja dolazi.

2.3.5. Implementacija protočnog deblokiranja

Rešavanje protočnih hazarda uvodjenjem hardverskih mehanizama naziva se **protočno-deblokiranje** (*pipeline interlock*). Hardver protočnog deblokiranja mora da detektuje sve protočne hazarde i obezbedi uslove kako bi zadovoljio sve zavisnosti.

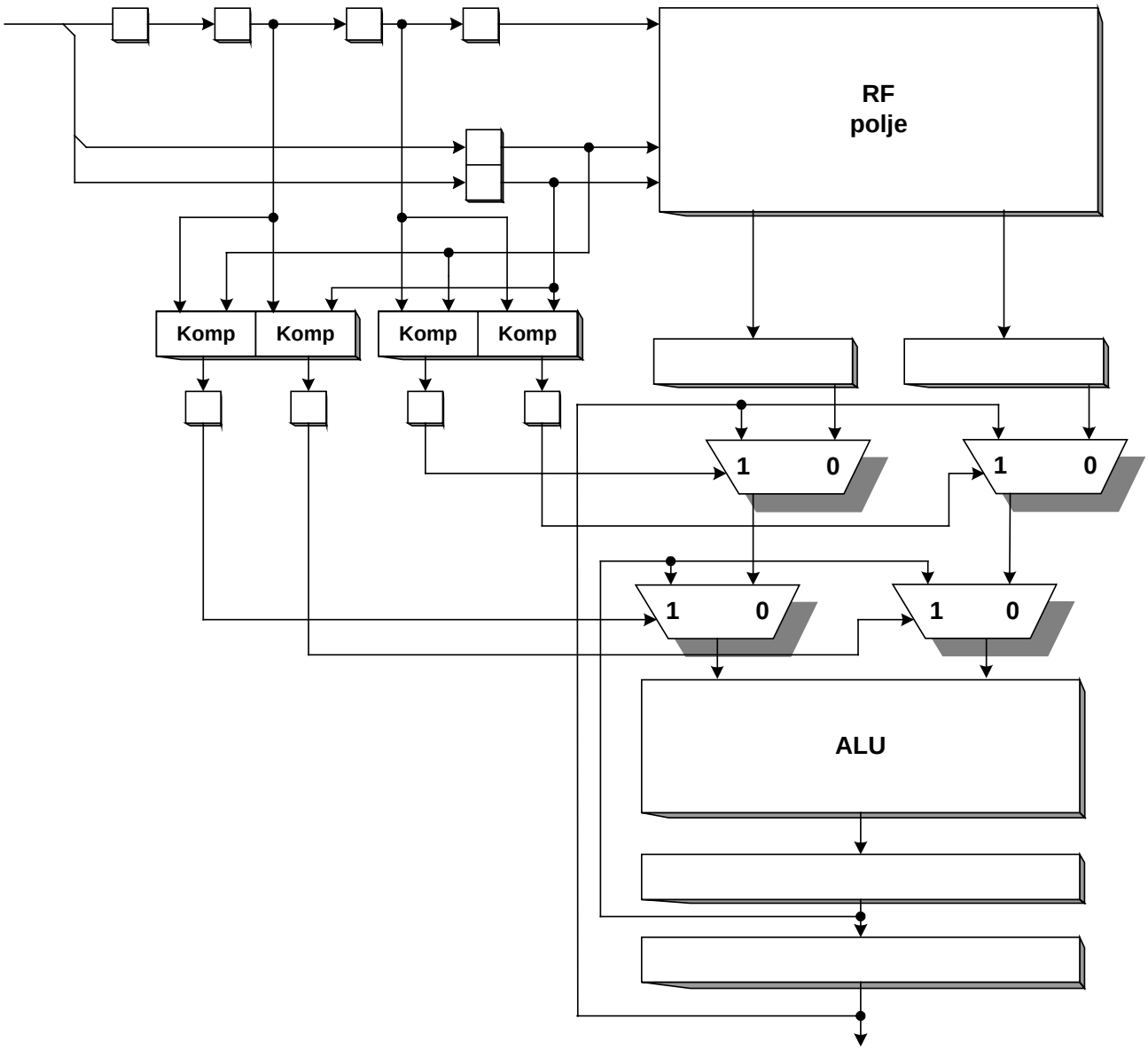
Uvodjenjem puteva za premošćavanje, skalarni protočni sistem ne može se više smatrati kao jednostavni linearno organizovan niz protočnih stepeni kod koga se podaci prenose od jednog stepena ka drugom. Putevi premošćavanja predstavljaju sada potencijalne povratne veze sa izlaza kasnijih stepena na ulaze stepena koji im prethode. Tako na primer, sva tri puta premošćavanja koja su neophodna da podrže ALU instrukciju u otklanjanju protočnog hazarda su prikazana na slici 2.17. Ove puteve nazivamo **ALU putevi premošćavanja** (*ALU forwarding paths*). Ispod slike 2.17 prikazano je kako je veći broj zavisnih pratećih instrukcija moguće zadovoljiti u toku naredna tri redosledna mašinska ciklusa. U toku ciklusa t_1, instrukcija *i* prosledjujesvoj rezultat zavisnoj instrukciji *i+1* preko puta za premošćavanje označen kao "*a*". U toku narednog ciklusa, t_2, instrukcija *i* prosledjuje svoj rezultat zavisnoj instrukciji *i+2* preko puta za premošćavanje "*b*". Ako instrukcija *i+2* zahteva rezultat od instrukcije *i+1*, ovaj rezultat se može takodje proslediti preko puta za premošćavanje " *a*" u toku ciklusa t_2. U toku ciklusa t_3, instrukcija *i* može da prosledi svoj rezultat instrukciji *i+3* preko puta za premošćavanje "*c*". Takodje, putevi "*a*" i "*b*" se mogu aktivirati u toku ciklusa t_3 ako instrukcija *i+3* zahteva rezultat instrukcija *i+2* i *i+1*, respektivno.



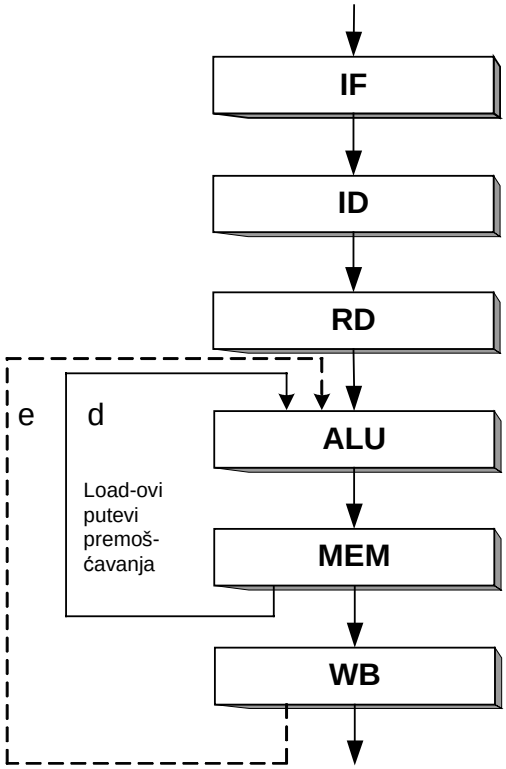
$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$	$i+3:\ \leftarrow\ R_1$
$i:\ R_1 \leftarrow$	$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$
	$i:\ R_1 \leftarrow$	$i+1:\ \leftarrow$
		$i:\ R_1 \leftarrow$
$(i \rightarrow i+1)$	$(i \rightarrow i+2)$	$(i \rightarrow i+3)$
premošćavanje preko puta "*a*"	premošćavanje preko puta "*b*"	i upisuje u R_1 pre nego što $i+3$ čita R_1

Slika 2.17 Putevi premošćavanja koji podržavaju protočne hazarde kada je vodeća instrukcija tipa ALU

Fizički, implementacija logičkog dijagrama sa slike 2.17 je prikazana na slici 2.18. Uočimo da se RAW hazardi detektuju koristeći komparatore koji uporedjuju specifikatore (adrese) registara uzastopnih instrukcija. Ako procesor TYP u *RF* polju ima ugradjeno 32 registra tada se ugradjuju četiri 5-bitna komparatora (vidi sliku 2.18). Ako se prateća instrukcija *j* tekuće nalazi u stepenu RD, tj. pokušava da pročita svoja dva registarska operanda, tada prva dva komparatora (na levoj strani slike) proveravaju da li postoje moguće RAW zavisnosti izmedju instrukcije *j* i *j-1*, koja se tekuće procesira u stepenu ALU. Ova dva komparatora uporedjuju oba specifikatora izvornih registara instrukcije *j* sa specifikatorom odredišnog registra instrukcije *j-1*. Istovremeno ostala dva komparatora (na desnoj strani) proveravaju da li postoje moguće RAW zavisnosti izmedju instrukcije *j* i instrukcije *j-2* koja se sada nalazi u stepenu MEM. Ova dva komparatora uporedjuju specifikatore oba izvorišna registra instrukcije *j* sa specifikatorom odredišnog registra instrukcije *j-2*. Izlazi sva četiri komparatora, u narednom ciklusu, se koriste kao upravljački signali radi aktiviranja odgovarajućih puteva za premošćavanje u slučaju kada se zavisnosti detektuju. Put premošćavanja "*a*" se aktivira od strane prvog para komparatora kada se detektuje(u) RAW zavisnost(i) izmedju instrukcije *j* i *j-1*. Na sličan način, put premošćavanja "*b*" se aktivira od strane izlaza drugog para komparatora kada se detektuje(u) zavisnost(i) izmedju instrukcija *j* i *j*-2. Oba puta za premošćavanje se mogu istovremeno aktivirati ako je instrukcija *j* zavisna od obe prethodne instrukcije, *j-1* i *j-2*.



Slika 2.18 Implementacija protočnog deblokiranja kod RAW hazarda kada je vodeća instrukcija tipa *ALU*



$i+1:\ \leftarrow\ R_1$	$i+1:\ \leftarrow\ R_1$	$i+2:\ \leftarrow\ R_1$
$i:\ R_1 \leftarrow\ MEM[\ ]$		$i+1:\ \leftarrow\ R_1$
	$i:\ R_1 \leftarrow\ MEM[\ ]$	
		$i:\ R_1 \leftarrow\ MEM[\ ]$
$(i \rightarrow i+1)$	$(i \rightarrow i+2)$	$(i \rightarrow i+3)$
zastoj $i+1$	premošćavanje preko puta "*d* "	i upisuje u R_1 pre nego što $i+2$ čita R_1

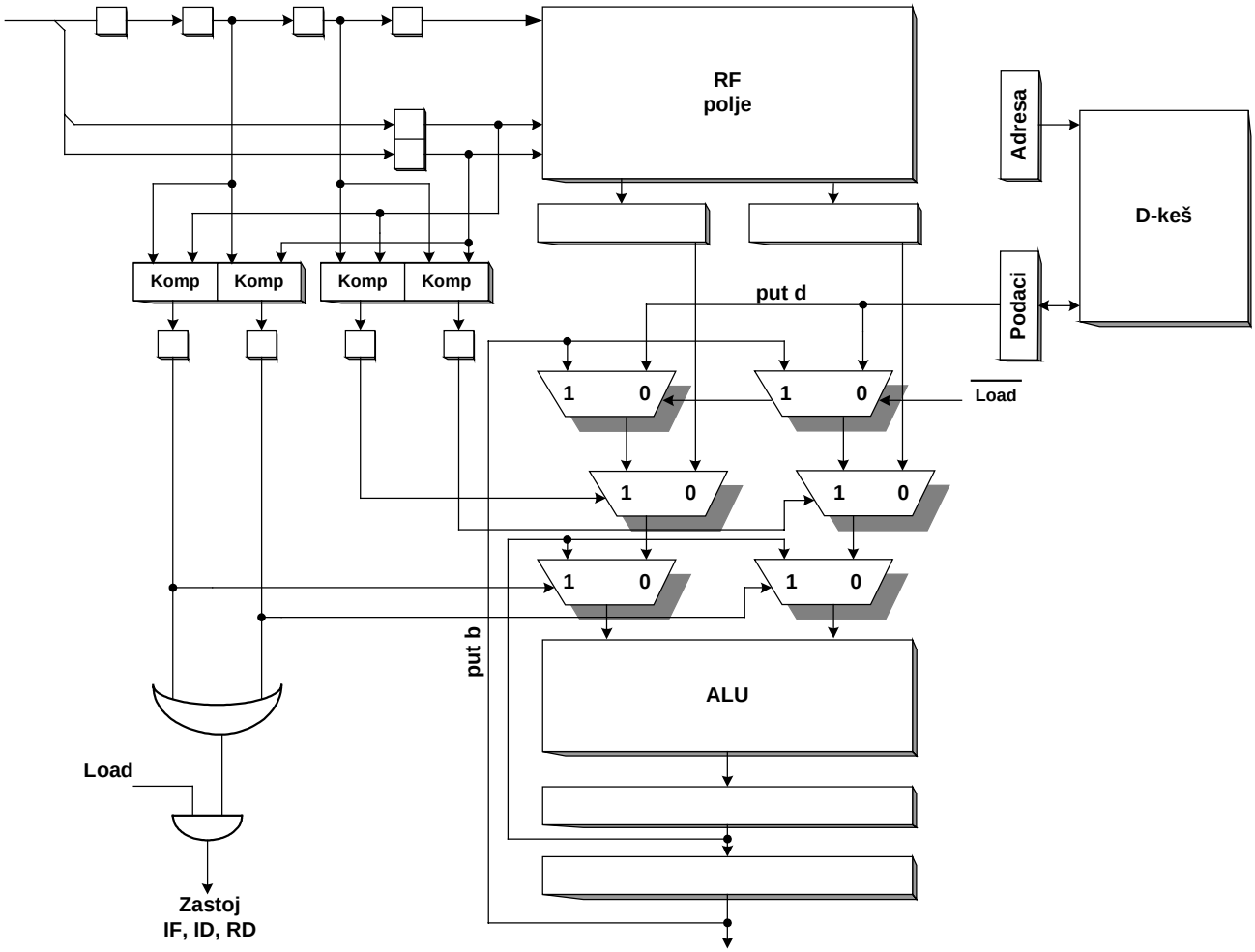
Slika 2.19 Putevi premošćavanja koji podržavaju protočne hazarde kada je vodeća instrukcija tipa *Load*

Put prosledjivanja "*c*" sa slike 2.17 nije prikazan na slici 21, a razlog je taj što ovaj put može biti nepotreban ako se preduzmu odgovarajuće akcije u toku projektovanja multi-portnog *RF* polja. Naime, ako se fizički dizajn tro-portnog *RF* polja (dva čitanja i jedan upis) ostvari tako da se u svakom ciklusu prvo obavi operacija upis, a nakon toga dve operacije čitanja, tada nema potrebe za trećim ("*c*") putem premošćavanja. U suštini instrukcija *j* pročitaće novu, korektnu, vrednost iz zavisnog registra u trenutku kada prolazi kroz stepen RD. To znači da se premošćavanje obavlja interno u okviru *RF* polja, pa shodno tome ne postoji potreba za čekanjem od jednog ili dva ciklusa, kako bi se pročitao sadržaj zavisnog registra ili da se njegova vrednost prosledi sa izlaza stepena WB na ulaz stepena ALU.

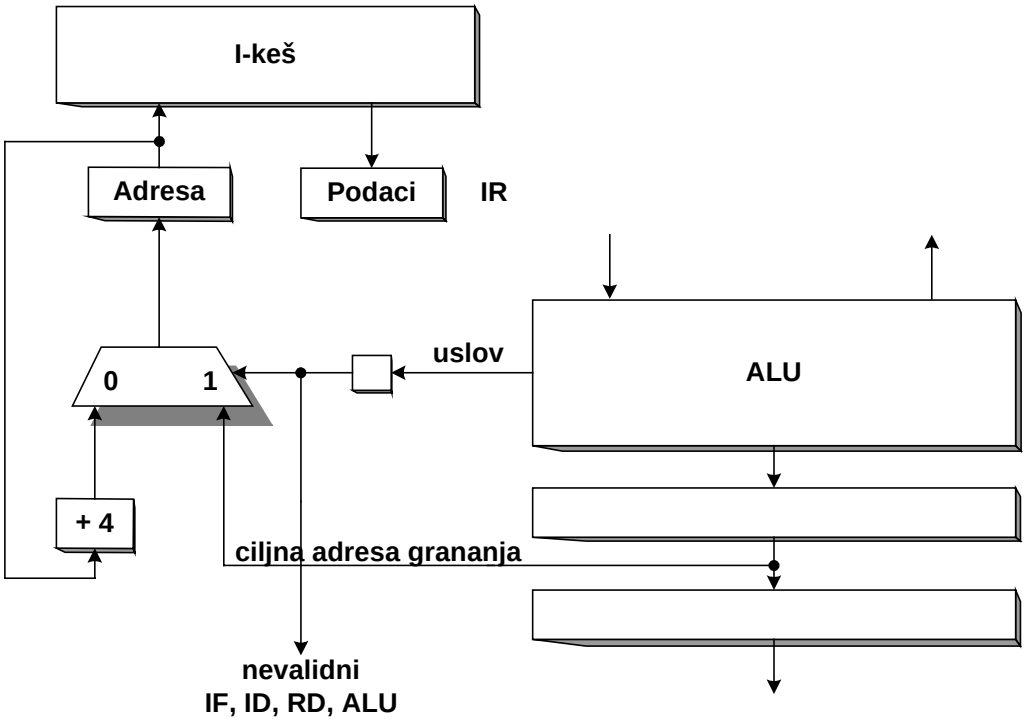
Da bi se redukovala cena koja se plaća zbog hazarda kada je vodeća instrukcija tipa *Load* neophodno je ugraditi dodatni skup puteva premošćavanja. Na slici 22 prikazana su dva puta premošćavanja koje je neophodno uvesti ako je vodeća instrukcija kod protočnog hazarda tipa *Load*. Ove puteve nazivamo **puteve premošćavanja** tipa **Load** (*Load forwarding paths*). Put "*d*" premošćava izlaz stepena MEM sa ulazom stepena ALU. Kada vodeća instrukcija pristigne do stepena ALU, ako je instrukcija *i+1* zavisna od instrukcije *i* tada ona mora da se zaustavi u stepenu RD u trajanju od jednog ciklusa. U narednom ciklusu kada instrukcija *i* napušta stepen MEM, njen rezultat se prosledjuje stepenu ALU preko puta "*d*" kako bi se obezbedili uslovi da instrukcija *i+1* udje u stepen ALU. Za slučaj kada i instrukcija *i+2* zavisi od instrukcije *i*, tada se isti rezultat prosledjuje i u narednom ciklusu preko puta "*e*" od stepena WB ka stepenu ALU, čime se obezbedjuje procesiranje instrukcije *i+2* u stepenu ALU bez umetanja drugog ciklusa zastoja. Ponovo, ako se u multi-portno *RF* polje prvo obavlja operacija upis, a nakon toga operacija čitanje, tada nema potrebe za uvodjenjem puta premošćavanja "*e*". Tako na primer, u istom ciklusu obaviće se prvo upis u registar *RF* polja u stepenu WB od strane instrukcije *i*, a zatim čitanje rezultata instrukcije *i* od strane instrukcije *i+2* koja se tekuće procesira u stepenu RD.

Fizička implementacija svih puteva premošćavanja koji podržavaju protočne hazarde kako od strane instrukcije tipa *ALU* tako i instrukcije *Load* prikazana je na slici 23. Put premošćavanje "*e*" nije prikazan jer je usvojeno da je *RF* polje tako projektovano da se prvo obavlja operacija upis, a zatim u istom ciklusu slede dve operacije čitanja. Uočimo da su ALU put prosedjivanja "*b*" kao i *Load* put prosledjivanja "*d*"(prikazani na slikama 2.13 i 2.14, respektivno, kao putevi koji prolaze sa izlaza stepena MEM a dolaze na ulaz stepena ALU) prikazani kao različiti fizički putevi (vidi sliku 23). Ova dva puta pobudjuju prvi par multipleksera, tako da se samo jedan od njih može selektovati, sve u zavisnosti od toga da li je vodeća instrukcija u MEM stepenu tipa *ALU* ili *Load*. Put premošćavanja "*b*" polazi od bafera u stepenu MEM u kome se čuva izlaz *ALU* operacije od prethodnog mašinskog ciklusa. Put premošćavanja "*d*" polazi od bafera u stepenu MEM u kome se čuva podatak koji je dobavljen iz D-keša.

Ista dva para komparatora se koriste za detekciju registarskih zavisnosti, nezavisno od toga da li je vodeća instrukcija tipa *ALU* ili *Load*. Dva para komparatora su neophodna jer hardver za deblokiranje mora da detektuje moguće zavisnosti izmedju instrukcija *i* i *i+1*, kao i izmedju instrukcija *i* i *i+2*. Ako je *RF* polje projektovano tako da se u svakom ciklusu prvo obavi operaciju upis, a zatim operacija čitanje, tada zavisnost izmedju instrukcija *i* i *i+3* je automatski ispunjena kada instrukcija *i* prolazi kroz stepen WB, a instrukcija *i+3* kroz stepen RD. Izlaz prvog para komparatora zajedno sa signalom iz stepena ID se koristi da ukaže da je vodeća instrukcija tipa *Load*. Na ovaj način generiše se kontrolni signal kojim se zaustavlja rad prva tri stepena protočnog sistema (IF, ID, RD) u trajanju od jednog ciklusa pod uslovom da je detektovana zavisnost izmedju instrukcija *i* i *i+1*, i da je instrukcija *i* tipa *Load*.



Slika 2.20 Implementacija protočnog deblokiranja kod RAW hazarda kada su vodeće instrukcije tipa *ALU i Load*

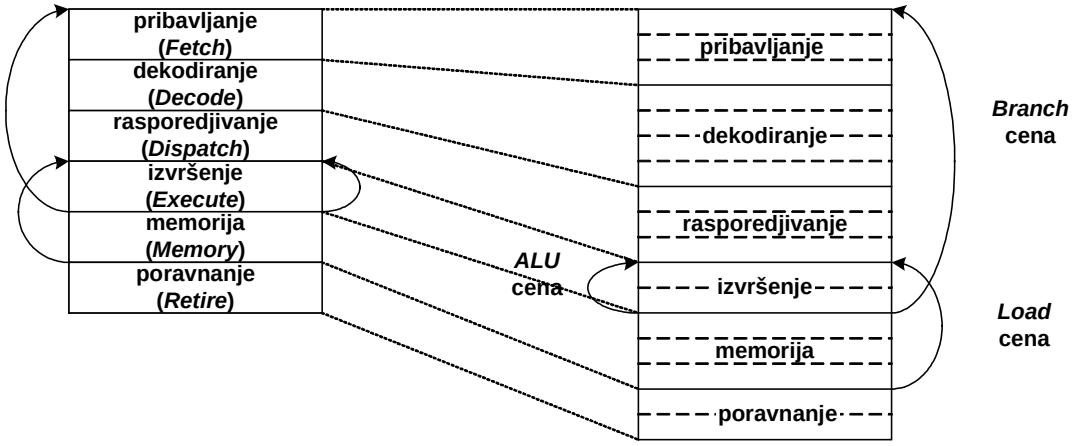


Slika 2.21 Implementacija protočnog deblokiranja kod hazarda koji se javljaju usled instrukcije *Branch*

Hardver za deblokiranje kod procesora TYP mora takodje uspešno da rešava i probleme koji nastaju kod protočnih hazarda iniciranih od strane upravljačkih zavisnosti. Implementacija mehanizma za deblokiranje koji podržava upravljačke hazarde, a bazira se na činjenici da je vodeća instrukcija tipa *Branch* prikazan je na slici 24. Normalno, u svakom ciklusu IF stepen pristupa I-kešu kako bi pribavio narednu instrukciju a istovremeno inkrementira PC da bi se pripremio za pribavljanje neredne sekvencijalne instrukcije. Kada se instrukcija tipa *Branch* pribavi od strane stepena IF, a zatim dekodira od strane stepena ID, rad IF stepena se može zaustaviti sve dok instrukcija *Branch* ne prodje kroz stepen ALU, u kome se generiše ciljna adresa grananja i izračunava uslov grananja. U narednom ciklusu, koji odgovara stepenu MEM instrukcije *Branch*, uslov grananja se koristi da napuni ciljnu adresu grananja u PC preko desne strane PC multipleksera, pod uslovom da *do-grananja-dodje*. Ovo rezultira cenom od četiri ciklusa zastoja, tj., cena koja mora da se plati kada se javi instrukcija grananja. Alternativno, može se usvojiti pristup da do grananja nikad ne dolazi, tako da IF stepen produžava da pribavlja naredne instrukcije sa sekvencijalnog programskog puta. U slučaju kada *do-grananja-dolazi*, PC se ažurira na ciljnu adresu u stepenu MEM dok instrukcije koje se nalaze u stepenima IF, ID, RD i ALU se invalidiraju i izbacuju iz protočnog sistema. U ovom slučaju, dolazi do zastoja od četiri ciklusa samo kada *do-grananja-dolazi*. Za slučaj da *do-grananja-ne-dolazi* cena zastoja iznosi nula ciklusa, tj. ne postoji zastoj u protočnoj obradi.

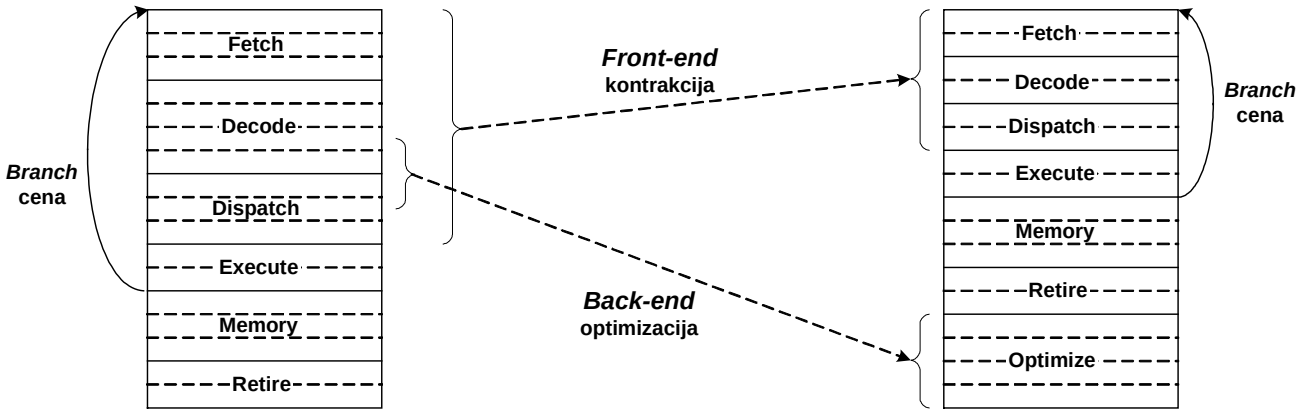
2.4. Procesori sa dubljom protočnošću

Dubljom protočnošću povećava se broj protočnih stepeni, ali se u svakom stepenu smanjuje broj nivoa ugradjenih logičkih kola. Dobra strana dublje protočnosti je mogućnost smanjenja vreme-trajanja mašinskog ciklusa, a shodno tome i povećanje taktne frekvencije. Tokom 80' godina prošlog veka protočni procesori su imali od 4 do 6 protočna stepena. Kod savremenih visoko-performansnih mikrorprocesora taktna frekvencija iznosi nekoliko GHz, dok je dubina protočne obrade veća od 2.17. Protočni procesori su postali ne samo dublji nego i širi, kakvi su superskalarni. No postoji i negativna strana dublje protočne obrade. Kako protočnost postaje dublja cena koja treba da se plati zbog postojanja protočnih hazarda postaje sve veća. Na slici 2.21 prikazano je šta se dešava sa cenom koja se plaća kod izvršenja *ALU, Load i Branch* instrukcija kada protočni sistem postaje širi i dublji. Uporedjivanjem plitke i duboke protočnosti vidimo da cena koja se plaća kod izvršenja *ALU* instrukcija iznosi jedan ciklus, kod instrukcija *Load* povećava se sa jedan na četiri ciklusa, a kod instrukcija tipa *Branch* povećava se sa tri na 11 ciklusa. U principu, kako se cena koja treba da se plati raste tako se povećava i prosečni *CPI*. Ipak da bi se zbog uvodjenja duboke protočne obrade ukupne performanse povećale neophodno je da uticaj (efekat) povećanja taktne frekvencije premaši povećanje *CPI*-a.



Slika 2.21 Uticaj cene koja plaća kod izvršenja instrukcija *ALU, Load i Branch* zbog povećanja dubine protočne obrade

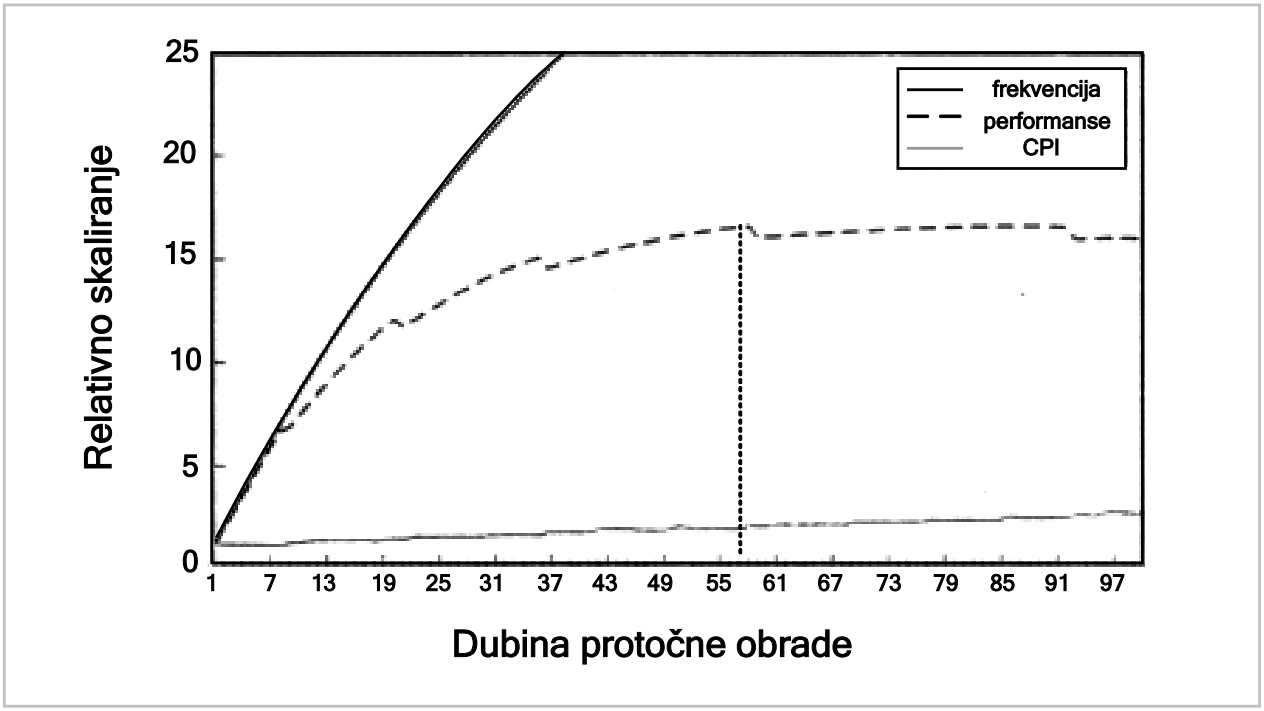
Postoje dva pristupa, koja se kod sistema sa dubljom protočnošću, standardno koriste za umanjenje negativnog efekta povećanja cene usled instrukcuje *Branch* (vidi sliku 2.22). Od svih povećanja cene, povećanje usled *Branch-a* je najveće jer ono zahvata (ima uticaj) na sve početne stepene u protočnoj obradi. Kad je predikcija grananja loša sve instrukcije koje se nalaze u početnim stepenima moraju da se isprazne (anulira njihov efekat). Prvi pristup da se smanji cena zbog grananja je da se smanji broj protočnih stepeni na početku obrade (u lancu obrade). Tako na primer, CISC arhitekturi koja koristi format instrukcija sa promenljivom dužinom potrebna je veoma složena logika za dekodiranje instrukcija, a to indirektno zahteva ugradnju većeg broja protočnih stepeni. Korišćenjem RISC arhitektura dekodiranje instrukcija postaje jednostavnije, a to rezultira manjem broju protočnih stepena koji se nalaze na početku lanca. Jedno alternativno rešenje se sastoji u korišćenju logike za pre-dekodiranje koja se ugradjuje pre punjenja instrukcije u I-keš. Pre-dekodiranim instrukcijama koje se pribavljaju iz I-keša poptrebno je daleko manje logike za dekodiranje pa otuda i manji broj dekoderskih protočnih stepeni.



Slika 2.22 Ublažavanje *Branch* cene kod duboke protočnosti

Drugi pristup se sastoji u premeštanju *front-end* kompleksnosti sa početka protočnog lanca na njegovom kraju (*back-end*). Kao rezultat dobijamo pliće *front-end* procesiranje, a shodno tome i manju *Branch* cenu.

Postoji još jedan interesantan aspekt koga treba razmotriti, a on se odnosi na to od kakvog je uticaja dubina protočnosti na *CPI*. U suštini tu treba da se pronadje kompromis izmedju povećanja taktne frekvencije i povećanja *CPI*-a. Na osnovu "*gvozdenog zakona*" performanse se odredjuju kao proizvod frekvencije i prosečnog *IPC*-a, tj. odnosa $\frac{frekvencija}{CPI}$. Kako protočni sistem postaje dublji, frekvencija se povećava, ali se takodje povećava i *CPI*. Povećanje dubine protočne obrade je opravdano sve dok imamo povećanje performansi. No postoji vrednost iznad koje dublja protočnost nema efekta. Pitanje koje se sada postavlja je sledeće: Koja je ta vrednost, tj. kolika duboka protočnost bi trebalo da bude ? Projektant Intel-a *Edward Grochowski* je vršio analizu koja se odnosi na skaliranje performansi sa povećanjem dubine protočnosti. Rezultati tih ispitivanja prikazani su na slici 2.23.



Slika 2.23 Skaliranje performansi sa povećanjem dubine protočne obrade

Kao što se vidi sa slike 2.23, performanse (izračunate kao $\frac{frekvencija}{CPI}$) imaju veoma brzi porast na početku, a zatim relativno mali porast kako se dubina protočnosti povećava. Na ekstremnom kraju krive sa povećanjem dubine protočne obrade evidentan je pad performansi. Analiza pokazuje da se poboljšanje performansi postiže sve dok je dubina protočne obrade manja od 57 stepeni, iznad te vrednosti performanse opadaju.

3. ORGANIZACIJA SUPERSKALARNIH PROCESORA

3.1. Ograničenja skalarnih protočnih procesora

Skalarni protočni sistemi se karakterišu jedinstvenim *k*-to stepenim protočno organizovanim sistemom za procesiranje instrukcija. Sve instrukcije nezavisno od tipa prolaze kroz isti skup protočnih stepena, ili kroz sve. Najviše po jedna instrukcija u datom trenutku može se naći u svakom protočnom stepenu, a instrukcije napreduju kroz protočne stepene jedna za drugom. Sa izuzetkom protočnih stepena koji su blokirani svaka instrukcija se procesira od strane protočnog stepena za tačno jedan ciklus i predaje se narednom stepenu u narednom ciklusu. Ovako stroga skalarna protočnost ima sledeća tri osnovna ograničenja o kojima ćemo u daljem tekstu detaljnije govoriti:

1. maksimalna propusnost skalarnog propusnog sistema iznosi (ograničena je) jednom instrukcijom po ciklusu
2. unifikacija svih tipova instrukcija u jedinstveni tip protočne obrade rezultira neefikasnom dizajnu
3. zaustavljanje rada protočnih stepeni dovodi do ubacivanja mehurova (*bublles*) u radu sistema.

Kao što smo već naglasili (vidi *gvozdeni zakon*) performanse procesora se mogu povećati povećanjem IPC-a i/ili frekvencije, ili smanjenjem ukupnog broja instrukcija.

$$Performance = \frac{1}{Instruction\ Count} \times \frac{Instructions}{Cycle} \times \frac{1}{Cycle\ Time} = \frac{IPC \times Frequency}{Instruction\ Count} \quad \text{..... 3.1}$$

gde su:

Performance - Performanse;

Instruction Count - Broj instrukcija;

$\frac{Instructions}{Cycle}$ - $\frac{\text{Instrukcije}}{\text{Ciklus}}$

$\frac{1}{Cycle\ Time}$ - Frekvencija

Frekvencija se može povećati korišćenjem dublje protočnosti. Dublja protočnost dovodi do manjeg broja nivoa logičkih gejtova u svakom protočnom stepenu (kraće vreme procesiranja po stepenu), a indirektno i do kraćeg vremena ciklusa (taktnog perioda) ili veće pobudne frekvencije. Realno postoji kritična tačka u povećanju dubine protočnosti. Naime kod dublje protočnosti cena koja treba da se plati (broj izgubljenih ciklusa) jako zavisi od zavisnosti izmedju instrukcija. Sa druge strane smanjenje vreme ciklusa može da ima negativni efekat na *CPI*.

Nezavisno od dubine protočnosti, skalarni protočni sistem može da inicira procesiranje jedne instrukcije svakog mašinskog ciklusa. U suštini, *IPC* kod skalarnog protočnog sistema je ograničen na jedan. Da bi se ostvarila veća propusnost, posebno ako dublja protočnost ne predstavlja neko ograničenje, sa aspekta cene, da bi se ostvarile bolje performanse, neophodno je inicirati više od jedne instrukcije u svakom mašinskom ciklusu. Ovo zahteva da se poveća širina protočnog sistema (obim reči koje se istovremeno procesiraju) kako bi se obezbedilo da se više od jedna instrukcija u datom

trenutku procesira od strane svakog protočnog stepena. Ovaj tip protočnog sistema nazvaćemo paralelni protočni sistem (*parallel pipeline*).

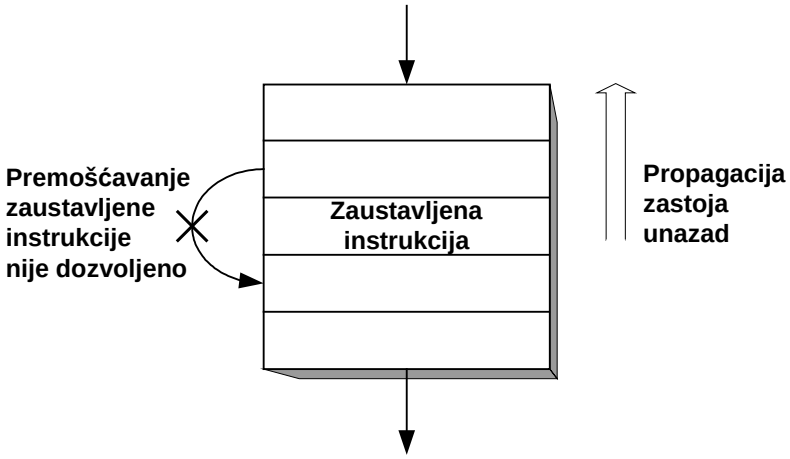
3.1.1. Neefikasna unifikacija u jedinstveni protočni sistem

Naglasimo da druga idealizovana pretpostavka o radu protočnog sistema svodi na to da se sva repetitivna izračunavanja procesiraju od strane protočnog sistema na jedinstveni način. Kod protočnog procesiranja instrukcije ovakva pretpostavka ne važi. Naime, postoje različiti tipovi instrukcija od kojih se svaka karakteriše svojim skupom izračunavanja. Unificiranje različitih zahteva u jedinstvenu protočnu obradu rezultira teškoćama i/ili neefikasnosti u radu sistema. Analizirajući rad protočnog sistema TYP zaključili smo da kod prvih stepena u protočnom lancu obrade (*IF*, *ID* i *RD*) postoji značajna uniformnost u radu ali na žalost, kod izvršnih stepeni (*ALU* i *MEM*) postoji izrazito neslaganje. U suštini, kod procesora TYP, ignorisali smo instrukcije tipa FP-*floating point* (ove instrukcije se više karakterišu kao CISC tipa jer značajno narušavaju koncept rada RISC procesora) kao i druge složene instrukcije za manipulisanje sa brojevima u fiksnom zarezu (kakve su množenje i deljenje) jer je za njihovo izvršenje potreban veći broj *EX*-ciklusa (ciklusa izvršenja). Instrukcije koje zahtevaju duže procesiranje i imaju promenljivu latentnost teško se unificiraju u poredjenju sa jednostavnim instrukcijama koje imaju kašnjenje od samo jednog latentnog ciklusa. Sa druge strane kako se disparitet u brzini rada izmedju CPU-a i memorije povećava, latentnost (kao broj mašinskih ciklusa) kod memorijskih instrukcija se permanentno povećava. Zbog razlika u latentnost hardverski resursi koji su neophodni da podrže izvršenje ovakvih različitih tipova instrukcija su takodje različiti. Konstantna presija da se realizuje što brži hardver rezultira realizacijom specijalizovanih izvršnih jedinica a to sa druge strane dovodi do značajne razlike u EX-stepenima kod protočnih sistema.

Shodno tome forsirana težnja za unificiranjem svih tipova instrukcija u jedinstvenu protočnu obradu, kod svih visoko-performansnih procesora, postaje takoreći nemoguća ili ekstremno neefikasna. Kod paralelnih protočnih sistema postoji stroga motivacija ne za unificiranjem izvršnog hardvera nego za implementacijom većeg broja različitih izvršnih jedinica (*EX-units*). Ovaj tip protočne obrade naziva se raznolika protočna obrada (*diversified pipelines*).

3.1.2. Gubitak perfomansi zbog stroge protočnosti

Skalarni procesori su rigidni u smislu da se advansiranje instrukcija kroz protočne stepene vrši jedna za drugom. Instrukcije ulaze u skalarni protočni sistem kako je to definisano redosledom u programu tj. *in order*. Kada ne postoje zastoji u radu protočnog sistema, sve instrukcije u protočnim stepenima advansiraju sinhrono tako da za programski redosled kažemo da je očuvan. Kada se neka od instrukcija zaustavi u neki protočni stepen zbog zavisnosti izmedju nje i neke instrukcije koja joj prethodi, tada se ta instrukcija zadržava u tom protočnom stepenu sve dok instrukcija koja joj prethodi ne završi procesiranje u stepenu gde je to potrebno da se obavi, tj. da se dobije rezultat. Zbog rigidne prirode u radu protočnog sistema, ako se zavisna instrukcija zaustavi u protočnom stepenu *i*, tada svi stepeni koji prethode (stepeni *1,2,....., i-1*) zadržavaju napredovanje narednih instrukcija. Svih *i* stepena u protočnom stepenu se zaustavlja sve dok instrukcija u stepenu *i* ne premosti unazad (preda) svoj zavisni operand. Nakon što je inter-instrukcijska zavisnost otklonjena, sve ostale *i* instrukcije mogu ponovo sinhrono da napreduju u protočnom sistemu. Kod rigidne skalarne protočnosti, zaustavljeni stepen na sredeni protočnog lanca, ima uticaj na sve stepene koje mu prethode, u suštini zaustavljanje stepena *i* se propagira prema nazad ka svim stepenima koji mu u protočnom lancu prethode.



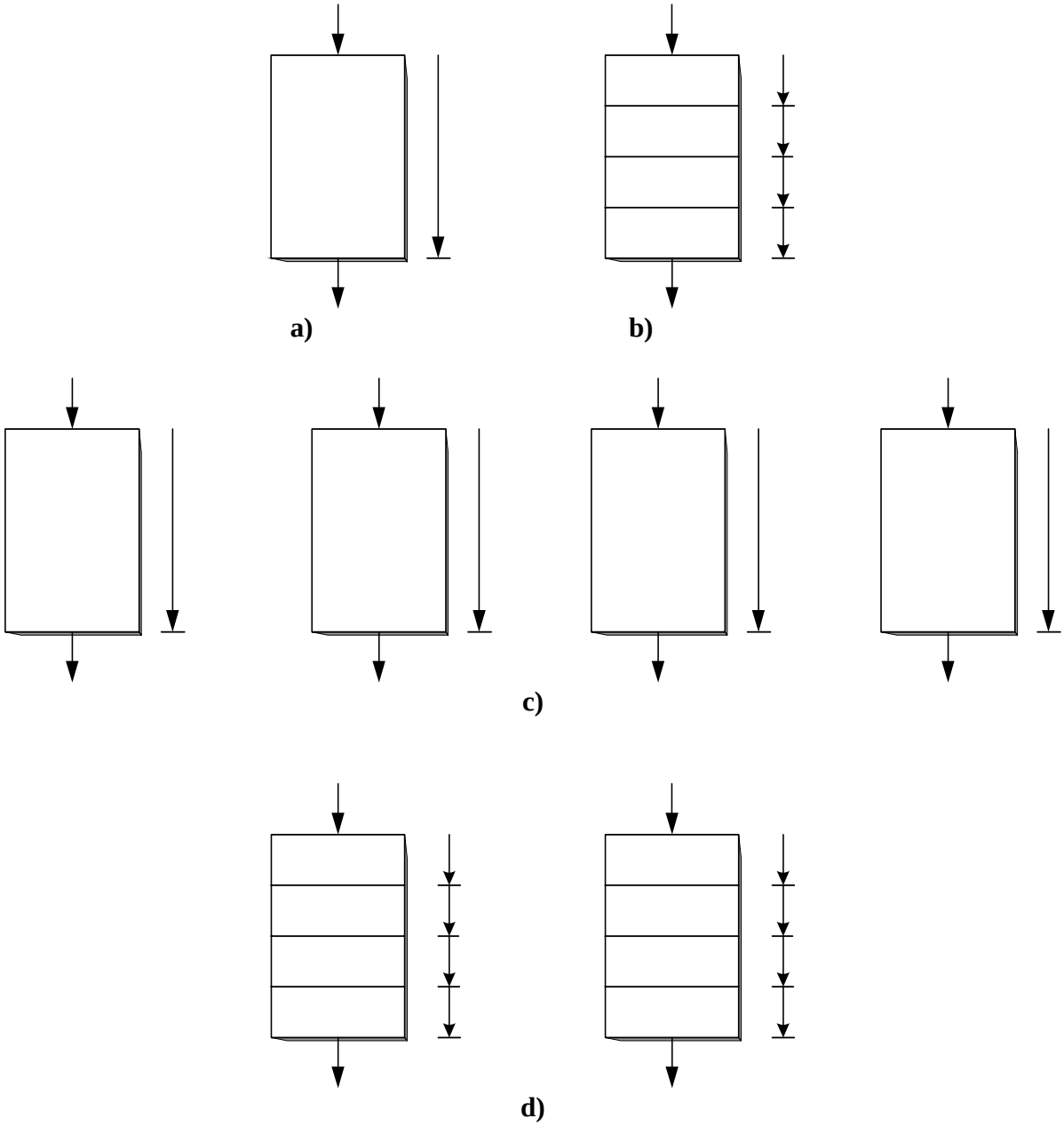
Slika 3.1. Propagacija zastoja unazad

Propagacija zastoja unazad od stepena koji je zaustavljen (*stalled*) (kod skalarnog protočnog sistema) uzrokuje pojavu mehurova (*bubbles*) u protočnoj obradi, ili tzv. zastoja u radu protočnog sistema. U toku rada skalarnog sistema može da se desi sledeći scenario. Neka od instrukcija se zaustavlja u stepenu *i* zbog zavisnosti koja postoji izmedju nje i instrukcije koja joj prethodi (recimo izmedju instrukcija *i* i *i-2*). No naglasimo pri tome da postoji i druga instrukcija, kakva je, u konkretnom slučaju, *i-1*, koja se takodje zaustavlja, a da pri tome zavisnosti izmedju instrukcije *i* i instrukcija *i-1* ne postoji. Saglasno programskoj semantici instrukcije *i-1* ne bi trebalo da se zaustavi. Ako je ovoj instrukciji ipak dozvoljeno premošćavanje i dalje napredovanje, zastoj u radu protočnog sitema se može efektivno redukovati tj. cena koja treba da se plati iznosi jedan ciklus (vidi sliku 3.1). Ako je većem broju instrukcija dozvoljeno da premoste instrukciju koja uzrokuje zastoj tada višestruka cena koja se plaća zbog zastoja može se eliminisati u tom smislu što se pasivni protočni stepeni koriste za procesiranje korisnih instrukcija. Omogućavanje premošćavanja vodećih instrukcija koje uzrokuju zastoj od strane pratećih instrukcija naziva se *van-redosledno-izvršenje* (*out-of-order-execution*) instrukcija. Rigidni skalarni protočni sistem ne dozvoljava *van-redosledno-izvršenje* pa shodno tome uzrokuje pojavu zastoja u radu koji se javljaju zbog zavisnosti izmedju instrukcija. Paralelni protočni sistemi koji podržavaju *van-redosledno-izvršenje* nazivaju se dinamički protočni sistemi (*dynamic pipelines*) .

3.2. Od skalarnih ka superskalarnim protočnim sistemima

Superskalarni protočni sistemi se mogu smatrati prirodnim naslednicima skalarnih protočnih sistema i predstavljaju poboljšanje u odnosu na skalarne sa tačke gledišta sledećih triju ograničenja:

1. Nasuprot skalarnim, superskalarni sistemi su paralelni protočni sistemi koji su u stanju da u svakom mašinskom ciklusu procesiraju veći broj instrukcija.
2. Kod superskalarnih mašina stepen *EX* se zasniva na korišćenju većeg broja heterogenih funkcionalnih jedinica (*Floating Unit*)-*FU* (jedna *FU* za *integer* sabiranje, druga za množenje, treća *FU* za *FP* sabiranje, itd.) dok kod skalarnih mašina *EX*-stepen se zasniva na jedinstvenoj funkcionalnoj jedinici (recimo *integer-ALU*).
3. Sa ciljem da se ostvare najbolje moguće performanse, a da se pri tome ne zahteva preuredjenje redosleda instrukcija od strane kompilatora (što je karakteristika *WLIV* mašina), superskalarni sistemi se mogu implementirati kao dinamički protočni sistemi.



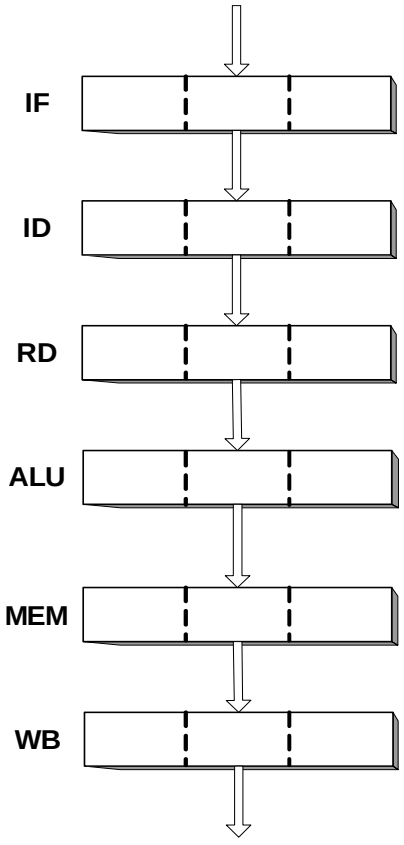
Slika 3.2 Paralelizam u radu mašine:a) ne postoji paralelizam (nema protočnosti); b) vremenski paralelizam (protočnost); c) prostorni paralelizam (veći broj funkcionalnih jedinica); d) kombinovani vremenski i prostorni paralelizam

3.2.1. Paralelna protočnost

Stepen paralelizma mašine se može meriti kao maksimalan broj instrukcija koje u datom trenutku mogu konkurentno da se izvršavaju. Kod *k*-to stepenog skalarnog protočnog sistema do *k* instrukcija može konkurentno biti rezidentno u mašini, pa je zbog toga njihovo potencijalno ubrzanje u odnosu na ne protočnu mašinu *k* puta. Alternativno, isti iznos ubrzanja se može postići korišćenjem *k* kopija ne protočne mašine radi paralelnog procesiranja *k* instrukcija. Ove dve forme mašinskog paralelizma su prikazane na slikama 3.2 b) i c), i one se mogu nazvati kao ***vremensko-mašinski-paralelizam*** (*temporal machine parallelism*) i ***prostorno-mašinski-paralelizam*** (*spatial machine parallelism*), respektivno. Vremenski i prostorni paralelizam istog stepena rezultiraju takoreći istom faktoru potencijalnog

ubzanja. No vremenski paralelizam koji se ostvaruje protočnom obradom zahteva ugradnju manjeg hardvera u odnosu na prostorni paralelizam, koji se bazira na principu repliciranja celokupne procesne jedinice. Paralelni protočni sistemi se mogu posmatrati kao sistemi koji koriste kako prostorni tako i vremenski paralelizam (vidi sliku 3.2 d)), pa shodno tome postižu veću protočnost u procesiranju instrukcija.

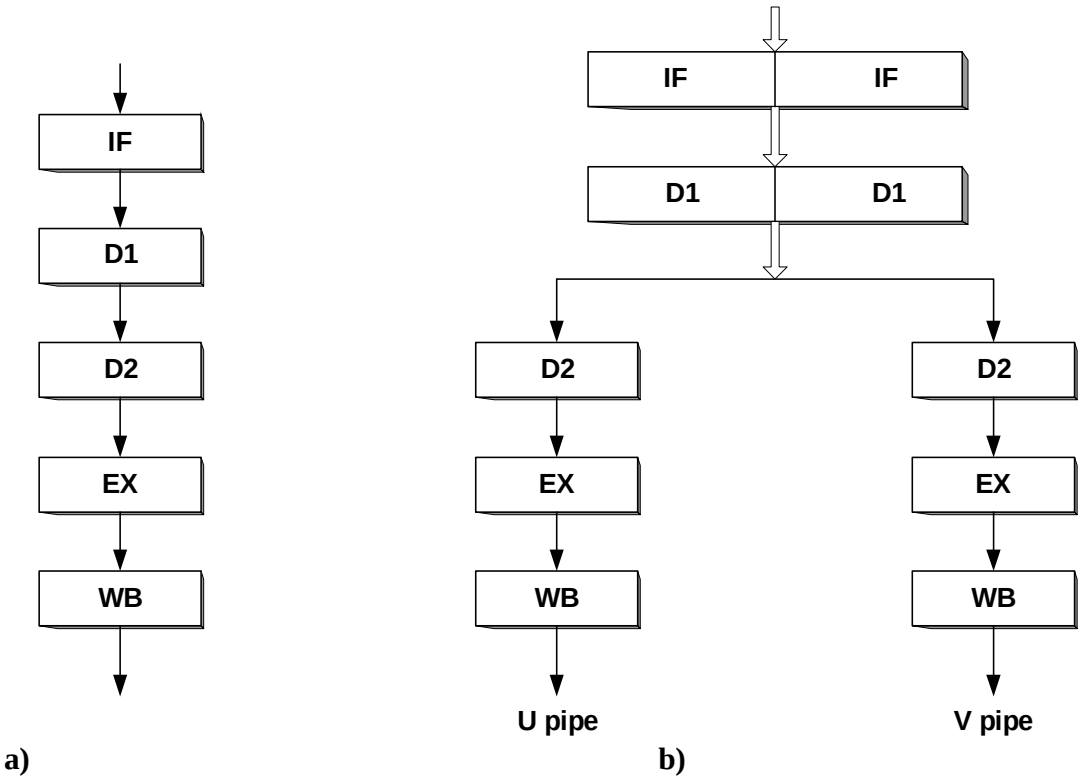
Ubrzanje skalarnog protočnog sistema se meri u odnosu na ne-protočni dizajn i prvenstveno je odredjeno od strane dubine skalarnog sistema. Kod paralelnih protočnih sistema, ili superskalarnih protočnih sistema, ubrzanje se obično meri u odnosu na skalarni protočni sistem i prvenstveno je odredjeno *obimom paralelne protočne obrade* (*width of the parallel pipeline*). Paralelni protočni sistem obima s može, u svakom od svojih protočnih stepeni, konkurentno da procesira do s instrukcija, što potencijalno predstavlja ubzanje od s u odnosu na skalarni sistem. Na slici 3.3 prikazan je paralelni protočni sistem obima $s = 3$.



Slika 3.3 Paralelni protočni sistem obima $s = 3$.

Za implementaciju paralelnog protočnog sistema neophodna je ugradnja značajnih hardverskih resursa. Svaki protočni sistem može potencijalno da procesira i advansira, u svakom mašinskom ciklusu, do s instrukcija. Shodno prethodnom logička kompleksnost svakog protočnog stepena se povećava za faktor s. U najgorem slučaju, kola za medjustepeno povezivanje se mogu povećati za faktor s^2, u slučaju da se koristi krozbar (*crosbar*) komutator obima $s \times s$ koji je u stanju da poveže svih s instrukcijskih bafera jednog stepena sa svim s instrukcijskim baferima narednog stepena. Sa ciljem da se podrži konkurentni pristup registrima *RF* polja od strane s instrukcija, broj portova za čitanje i upis u *RF* polju mora da se poveća za faktor s. Na sličan način, moraju se ugraditi i dodatni portovi kako za pristup I-kešu tako i za pristup D-kešu.

Kao što smo prethodno, u Poglavlju 2, naglasili mikroprocesor *Intel i486* je u suštini bio petostepeni skalarni protočni sistem. Naredni mikroprocesor iz familije *Intel* je bio *Pentium*. Mikroprocesor *Pentium* je superskalarna mašina koja implementira paralelnu protočnost obima $s = 2$. U suštini *Pentium* implementira dve *i486* mašine vidi sliku 3.4. Veći broj instrukcija, u svakom mašinskom ciklusu, se mogu pribavljati i dekodirati od strane prva dva paralelna protočna stepena. U svakom ciklusu, moguće je u EX-stepenima potencijalno inicirati izvršenje po dve instrukcije *EX*–stepene nazivamo *U* i *V* . Cilj je pri ovome da se ostvari iniciranje izvršenja od dve instrukcije po ciklusu.



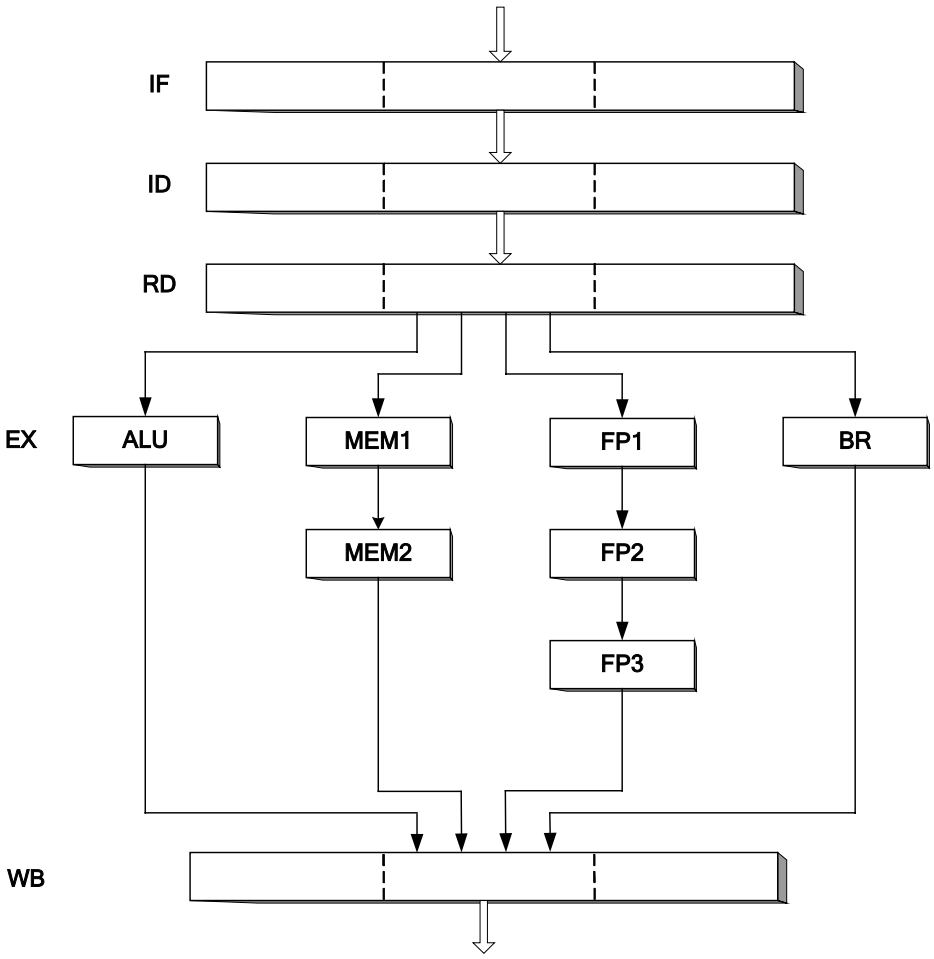
Slika 3.4 a) 5-stepeni skalarni protočni sistem *i486*; b) 5-stepeni paralelni protočni sistem *Pentium* obima *s=2*

U odnosu na skalarni protočni sistem *i486*, kod *Pentium* paralelnog protočnog sistema neophodno je ugraditi značajno veći broj hardverskih resursa. Kao prvo, 5-stepeni protočni sistem treba da se duplira po obimu. Oba protočna *EX*-stepena, mogu u svakom od svoja tri stepena da prihvataju po dve instrukcije. *EX*-stepen može da obavlja *ALU* operaciju ili da pristupa D-kešu. Shodno prethodnom neophodno je obezbediti dodatne portove kod *RF polja* kako bi se ostvarilo konkurentno izvršenje od dve operacije po ciklusu. Ako su obe instrukcije koje se izvršavaju u stepenu *EX* tipa *Load/Store* tada i D-keš mora da obezbedi dva pristupa. Pravi D-keš tipa *dual-port* je skup za implementaciju. Umesto toga D-keš kod *Pentium*-a se implementira kao *single-port* D-keš sa 8-strukim *interleaving*-om. Na ovaj način, podržava se simultani pristup dvema različitim bankama od strane *Load/Store* instrukcija koje se procesiraju u *U* i *V* stepenima. Ako postoji konflikt kod pristupa banci, tj. obe *Load/Store* instrukcije mora da pristupe istoj banci, tada se oba pristupa D-kešu moraju serijalizovati.

3.2.2. Raznovrsne protočne obrade

Hardverski resursi koji su potrebni za podršku radu izvršenja različitih tipova instrukcija mogu značajno da se razlikuju. Kod skalarnih procesora svi raznovrsni zahtevi koji se odnose na izvršenje svih tipova instrukcija moraju biti ujedinjeni (unificirani) u jedinstveni protočni sistem. Zbog toga protočna obrada može da bude veoma neefikasna, iz razloga što svaki tip instrukcije zahteva samo podskup izvršnih stepeni, ali ta instrukcija mora da prodje kroz sve izvršne stepene. Zbog toga svaka instrukcija je neaktivna kada prolazi kroz nepotrebne stepene i uzročnik je značajne dinamičke spoljne fregmentacije. Latencija izvršenja svih tipova instrukcija je ista i jednaka je ukupnom broju izvršnih stepeni. Ovakav princip rada dovodi do nepotrebnih zastoja pratećih instrukcija i/ili zahteva uvodjenje dodatnih puteva za premošćavanje.

Neefikasnost zbog unifikacije u jedinstveni protočni sistem kod paralelnih protočnih sistema se rešava korišćenjem različitih funkcionalnih jedinica u *EX* stepenu(ima). Umesto da se imlementiraju *s* identičnih protočnih stepena od kojih je svaki istog obima *EX* deo protočnog sistema se implementira pomoću razrnorodnih protočnih stepeni (vidi sliku 3.5). Kod ovog primera implementirana su 4 izvršna protočna stepena, ili funkcionalne jedinice, različitih protočnih dubina. Stepen RD, u zavisnosti od tipa instrukcije, rasporedjuje (dodeljuje) instrukcije svim izvršnim protočnim stepenima.

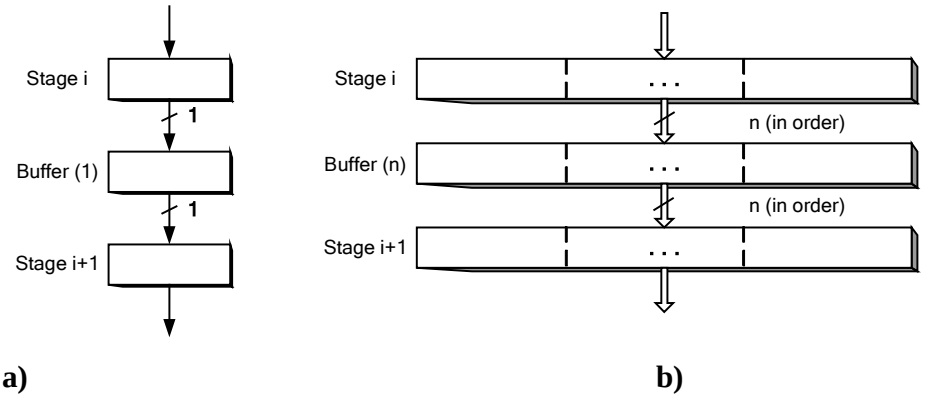


Slika 3.5 Raznolika paralelna protočna mašina sa četiri *execution* stanja

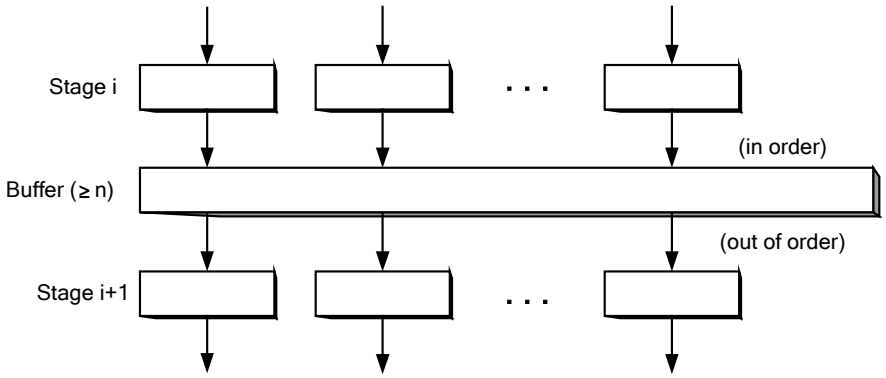
Kod projektovanja raznovrsnog paralelnog protočnog sistema potrebno je obratiti posebnu pažnju. Jedan važan aspekt se odnosi na to koliki broj i kakav tip funkcionalnih jedinica treba ugraditi. U idealnom slučaju broj funkcionalnih jedinica treba da bude uskladjen sa *ILP*-om programa, a tipovi funkcionalnih jedinica mora da su uskladjeni sa raznorodnim tipovima instrukcija koje postoje u programu. Kod prvih generacija superskalarnih procesora ugradjivala se druga funkcionalna jedinica koja se koristila sa manipulisanje brojevima u pokretnom zarezu tipa *floating point*, dok je prva bila namenjena sa manipulisanje brojevima tipa *integer*. Evolucija dizajna superskalarnih procesora kasnije je rezultirala *two-issue* i *four- issue* mašinama (mašine koje su u stanju da inicirajui zvršenje od dve i četiri instrukcije, respektivno). Kod ovih mašina implementiraju se po četiri funkcionalne jedinice od kojih se jedna koristi za manipulisanje sa *integer* vrednostima, druga sa *floating-point* brojevima, treća sa *Load/Store* instrukcijama, a četvra sa *Branch* instrukcijama. Neka novija rešenja inkorporiraju veći broj *integer* jedinica koje su namenjene za izvršenje *integer* operacija duže latentnosti kakve su operacije tipa množenje, deljenje, *add-multiply* ili drugih specifičnih operacija koje se često koriste u signal procesiranju, računarskoj grafici itd.

3.2.3. Dinamička protočna obrada

Kod svakog protočnog sistema ili dizajna neophodno je ugraditi bafere izmedju protočnih stepena. Kod strogo skalarnog protočnog sistema, kako je prikazano na slici 3.8 izmedju dva uzastopna protočna stepena (stepeni *i* i *i+1*) postavlja se bafer dubine jedan (u smislu instrukcija i čini ga jedan *n*-to bitni registar). U ovom baferu se čuvaju svi esencijalni upravljački bitovi instrukcije koja se tekuće procesira od strane stepena *i* , zatim upravljački bitovi te instrukcije koja će u narednom mašinskom ciklusu da se procesira u stepenu *i+1*, itd. Baferom dubine jedan (*single entry buffer-SEB*) je relativno lako upravljati. Svakog mašinskog ciklusa tekući sadržaj bafera se koristi kao ulaz u naredni protočni stepen *i+1*, a na kraju ciklusa bafer lečuje podatke generisane od strane prethodnog stepena *i*. To znači da se bafer taktuje svakog mašinskog ciklusa. Izuzetak se javlja kada instrukcija u baferu mora da se zaustavi i zabrani njen prolaz ka stepenu *i+1*. U tom slučaju, taktovanje bafera nije dozvoljeno i napredovanje instrukcija se zaustavlja u baferu. To znači da ako se neki od bafera u strogo protočnom skalarnom procesoru zaustavi, tada svi stepeni koji prethode stepenu *i* moraju takodje da se zaustave. Za slučaj kada ne postoje zastoji tada se svaka instrukcija zadržava u svakom baferu samo po jedan mašinski ciklus, a zatim napreduje prema narednom baferu. To znači da sve instrukcije ulaze i napuštaju bafer u istom redosledu kao što je to specificirano u sekvencijalnom programu.



a) **b)**



c)

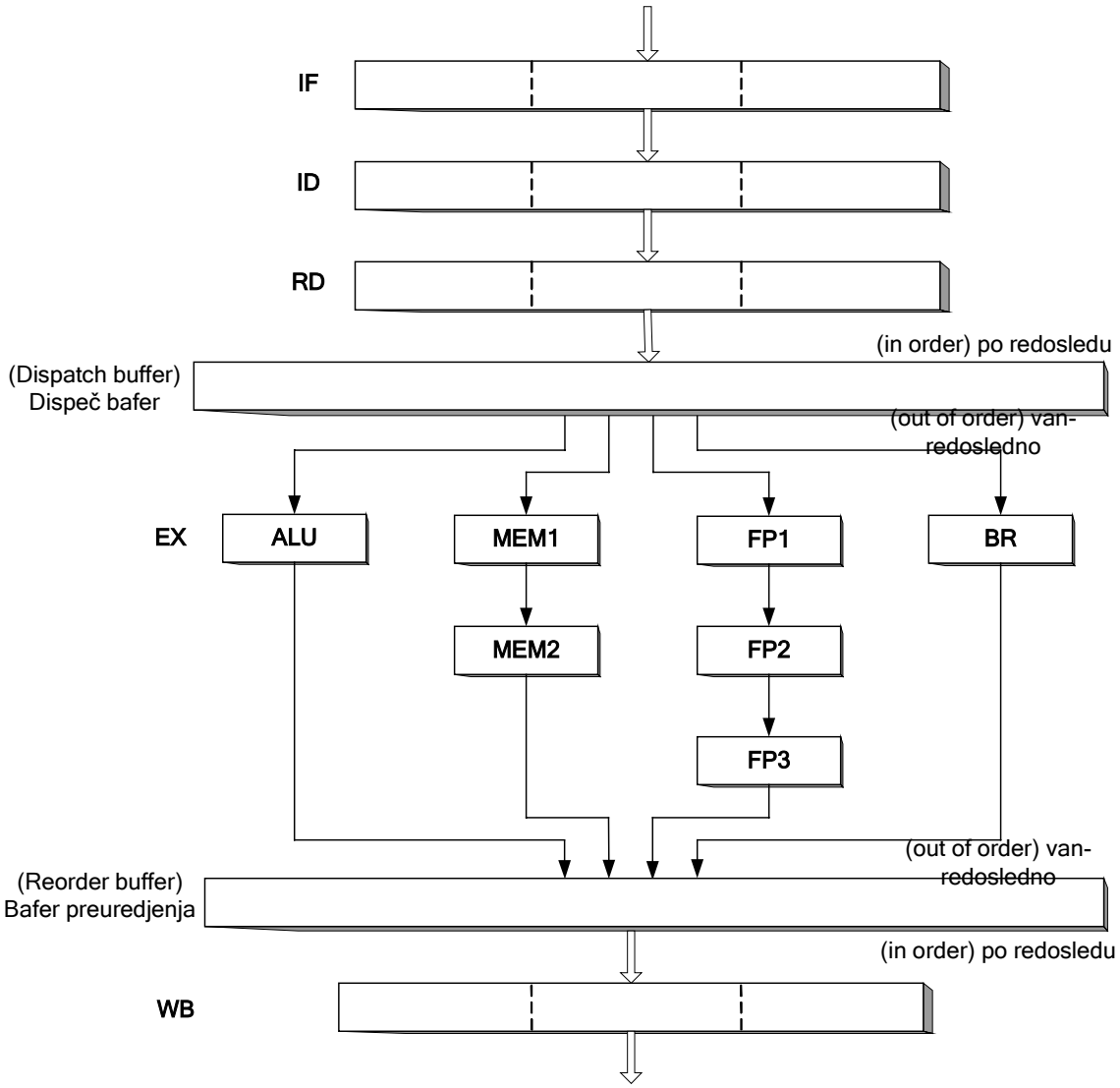
Slika 3.6 Baferi locirani izmedju protočnih stepena: a) bafer dubine jedan (*SEB*); b) više-ulazni bafer (*MEB*); c) više-ulazni bafer sa preuredjenjem

Kod paralelnog protočnog sistema, kako je to prikazano na slici 3.6 b), izmedju dva uzastopna protočna stepena ugradjuju se baferi sa većim brojem ulaza (*multientry buffer-MEB*). Ovi baferi se mogu posmatrati kao prosto proširenje bafera sa jednostrukim ulazom. Svakog mašinskog ciklusa u svakom od *MEB* može se lečovati veći broj instrukcija. U narednom ciklusu ove instrukcije mogu da prolaze kroz naredni protočni stepen. Ako sve instrukcije u *MEB*-u treba da advansiraju simultano (napreduju frontalno tj. da jedna ne ide ispred druge) tada upravljanje radom sa *MEB* je slično kao ono koje se odnosi na *SEB*. Naime *MEB* se taktuje ili zaustavlja u svakom mašinskom ciklusu isto kao *SEB*. Ipak, kod paralelnog protočog sistema, u toku rada, može da se desi sledeći scenario: Bilo koja od instrukcija može da dovede do nepotrebnog zastoja neke ili svih od instrukcija kod *MEB*-a. Zbog toga, radi ostvarivanja efikasnijeg rada paralelnog protočnog sistema sofisticiranija rešenja *MEB*-a su neophodna.

Svaki ulaz jednostavnog *MEB*-a sa slike 3.6 d) je direktno povezan na jedan *write port* i *read port*, pri čemu ne postoji interakcija izmedju ulaza bafera. Jedno poboljšanje u odnosu na jednostavni bafer sastoji se u uvodjenju medju-povezivanja izmedju ulaza sa ciljem da se olakša premeštanje podataka izmedju ulaza. Na primer ulazi se mogu povezati u linearni lanac kakav je pomerački registar i da funkcionišu na principu reda čekanja tipa *FIFO*. Jednim drugim poboljšanjem obezbedjuje se mehanizam za nezavisni pristup svakom ulazu bafera. Ovakvo rešenje iziskuje da postoji mogućnost eksplicitnog adresiranja svakog individualnog ulaza u bafer, kao i ne zavisno upravljanje operacijama *Read/Write* nad svakim od ulaza. Ako je svakom ulazno/izlaznom portu bafera obezbedjena mogućnost da pripada bilo kom ulazu bafera, tada će se jedan takav višeulazni bafer (*MEB*) ponašati kao multiportni RAM malog kapaciteta. Sa jednim takvim baferom instrukcija se može zadržati na ulaz u bafer za veći broj mašinskih ciklusa, i može se ažurirati ili modifikovati ako je rezidentna u tom baferu. Dalja poboljšanja mogu da se ostvare inkorporiranjem asocijativnog pristupa koji se odnosi na ulaze bafera. Umesto da se koristi konvencionalno adresiranje radi indeksiranja ulaza u bafer, sadržaj bilo kog ulaza se može koristiti kao asocijativni marker (*tag*) radi indeksiranja tog ulaza. Sa ovakvim mehanizmom pristupa, višeulazni bafer (*MEB*) se ponaša kao asocijativna keš memorija malog kapaciteta.

Superskalarni protočni sistemi razlikuju se u odnosu na (rigidno) skalarne protočne sisteme po jednom ključnom aspektu koji se odnosi na korišćenje složenih višeulaznih bafera radi baferovanja instrukcija koje se " u letu " izvršavaju. Sa ciljem da se minimiziraju nepotrebni zastoji instrukcija kroz paralelni protočni sistem, pratećim instrukcijama mora da se dozvoli da premoste vodeće instrukcije čije je napredovanje zaustavljeno. Ovakvo premošćavanje može da promeni redosled izvršenja instrukcija u odnosu na prvobitni sekvencijalni redosled definisan statičkim kôdom. Kod *van-*

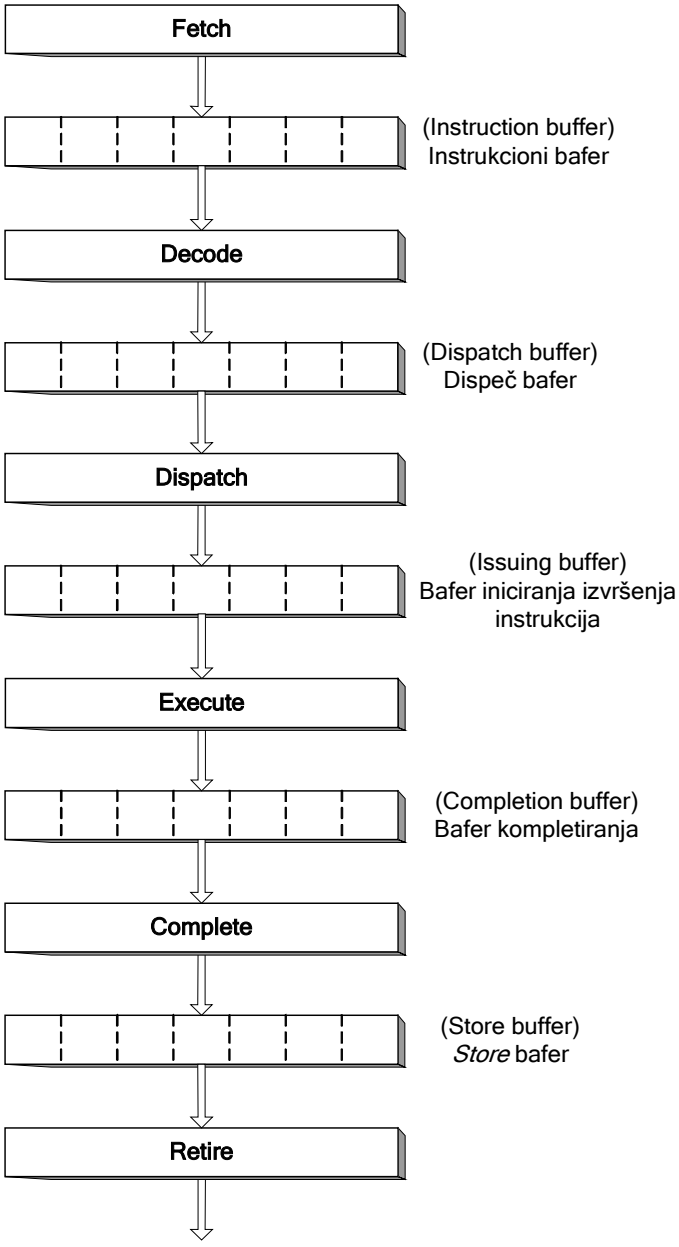
redoslednog (*out-of-order*) izvršenja instrukcija postoji potencijal da se približimo granici propusnosti koji se odnosi na tok prenosa podataka, a važi za izvršenje instrukcija, tj. instrukcije počinju sa izvršenjem odmah nakon dostupnosti njihovog operanda. Paralelni protočni sistem koji podržava *van-redosledno* izvršenje instrukcija naziva se dinamički protočni sistem (*dynamic pipeline*). Dinamički protočni sistem ostvaruje *van-redosledno* izvršenje zahvaljujući korišćenju složenih višeulaznih bafera koja omogućavaju da instrukcije ulaze i napuštaju bafere u različitim redosledima. Ovakav jedan višeulazni bafer koji ima mogućnost da ostvari preuredjenje redosleda prikazan je na slici 3.6 c). Slika 3.7 prikazuje jedan paralelni raznovrsni protočni sitem čiji je stepen superskalrnosti $s = 3$ i koji je organizovan kao dinamički protočni sistem.



Slika 3.7. Dinamički protočni sistem stepena superskalarnosti $s = 3$

Izvršni deo (*EX*) protočnog sistema se sastoji od 4-protočno organizovane funkcionalne jedinice, pri čemu je stepen *EX* ogradjen od strane dva višeulazna bafera koji omogućavaju da se ostavari preuredjenje redosleda instrukcija. Prvi bafer koji se naziva dispečer bafer (*dispatch buffer*) se puni dekodiranim instrukcijama u saglasnosti sa programskim redosledom, a zatim rasporedjuje instrukcije ka funkcionalnim jedinicama u redosledu koji je različit od programskog redosleda. Saglasno tome instrukcije mogu da napuste dispečer bafer u različitom redosledu u odnosu na redosled

kojim su one ušle u dispečer bafer. Ovaj protočni sistem takodje implementira skup raznovrsnih funkcinalnih jedinica od kojih svaka ima različitu latenciju. Mogućnost da se u funkcionalnim jedinicama inicira (*issuing*) *van-redosledno izvršenje* instrukcija, pri čemu mogu da imaju promenljivu latentnost ukazuje na to da instrukcije mogu da završavaju *van-redosleda*. Da bi obezbedili mehanizam koji garantuje da će izuzeci biti obradjeni u saglasnosti sa originalnim programskim redosledom, instrukcije moraju biti kompletirane (*completed*), tj. da ažuriraju mašinsko stanje u programskom redosledu. Sa *van-redoslednim* završetkom instrukcija, ostvarivanje redoslednog završetka (*completion*), postiže se ugradnjom drugog višeulaznog bafera koji je lociran na *back-end*-u izvršnog dela (*EX*) protočnog sistema. Ovaj bafer se naziva *completion buffer* (vidi sliku 3.10). i koristi se za baferovanje instrukcija koje mogu da završe *van-redosledno*, a izvlače (*retires*) se iz njega u programskom redosledu nakon čega se upisuju u konačni *WriteBack* stepen. Ovako dinamičko organizovani protočni sistem olakšava *van-redosledno* izvršenje instrukcije sa ciljem da postigne najkraće moguće vreme izvršenja, a što je takodje važno da obezbedi preciznu obradu izuzetaka izvlačenjem instrukcija iz *completion buffer*-a i ažuriranjem stanja mašine u saglasnosti sa programskim redosledom.



Slika 3.8 Šestostepeni TEMPLATE (TEM) superskalarni protočni sistem

3.3. Pregled principa rada superskalarnih protočnih mašina

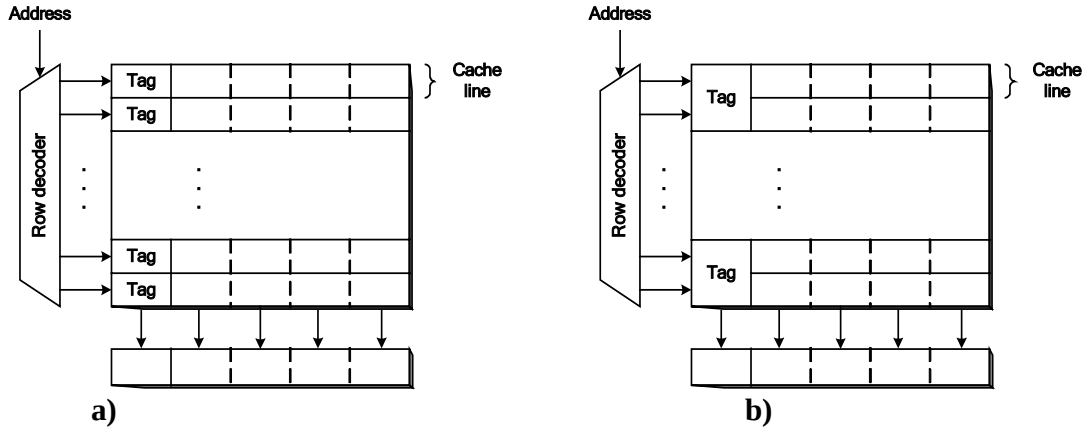
Ukazaćemo sada na ključne stavke koje se odnose na projektovanje superskalarnih protočnih mašina (*SSPM*). Našu pažnju usmerićemo, pre svega, ka opisu organizacije, ili strukturnom dizajnu *SSPM*-a. Tehnike i metode rada koje se odnose na dinamičku interakciju organizacije mašine, semantiku instrukcija, kao i optimizaciju performansi mašina biće date u narednom poglavlju.

Na sličan način kao i sa šestostepenim TYP protočnim sistemom (objašnjen u poglavlju 2 koji se odnosi na skalarni protočni dizajn) u ovom slučaju koristićemo šestostepeni TEM superskalarni protočni sistem prikazan na slici 3.8 kao šablon za dalju diskusiju u vezi organizacije *SSPM*-a. U odnosu na skalarne protočne sisteme postoje značajne razlike u implementaciji. Naglasimo da TEM *SSPM* ne treba da se posmatra kao neka aktuelna implementacija neke tipične ili reprezentativne *SSPM*. Šest stepena kod TEM *SSPM*-a treba gledati kao jedan logički protočni sistem koji može, ali ne mora, da odgovara nekom šestostepenom fizičkom sistemu. No sva šest stepena kod TEM *SSPM*-a pružaju jedan solidan okvir za diskusiju glavnih gradivnih blokova kod najvećeg broja *SSPM*-a.

Šest stepeni kod *SSPM*-a su: *Fetch, Decode, Dispatch, Execute, Complete* i *Retire*. Stepen *Execute* može da sadrži veći broj (protočno organizovanih) funkcionalnih jedinica različitih tipova i različite izvršne latentnosti. Ovo iziskuje da stepen *Dispatch* distribuira instrukcije različitih tipova ka odgovarajućim funkcionalnim jedinicama. Kod *van-redoslednog* izvršenja instrukcija u stepenu *Execute*, stepen *Complete* treba da obavi preuredjenje redosleda instrukcija i obezbedi (ostvari) redosledno ažuriranje stanja mašine. Ukažimo takodje da postoje višeulazni baferi (*MEB*) koji se koriste za razdvajanje ovih šest stepeni. Složenost ovih bafera može da varira u zavisnosti od njihove funkcionalnosti i lokacije u *SSPM*-u. Sagledaćemo sada ulogu svakog od ovih šest stepena.

3.3.1. Pribavljanje instrukcija

Nasuprot skalarnoj protočnoj mašini (*SPM*) za *SSPM* kažemo da je paralelna mašina koja je u stanju da svakog mašinskog ciklusa iz I-keša pribavlja više od jedne instrukcije. Za datu *SSPM* dubine *s* protočni stepen *Fetch* treba svakog mašinskog ciklusa da pribavi *s* instrukcija iz I-keša. To znači da fizička organizacija I-keša mora biti dovoljno široka i da svaka vrsta I-keša pamti *s* instrukcija, a da se svakoj vrsti pristupa jednovremeno. U daljoj diskusiji usvojićemo da je latencija pristupa I-keša kao i broj pribavljenih instrukcija jednak obimu vrste. Obično kod ovakve keš organizacije keš linija odgovara fizičkoj vrsti u keš polju, ali takodje je mogući slučaj da se keš linija bude raspodeljena u nekoliko fizičkih vsta keš polja, kako je to prikazano na slici 3.9.



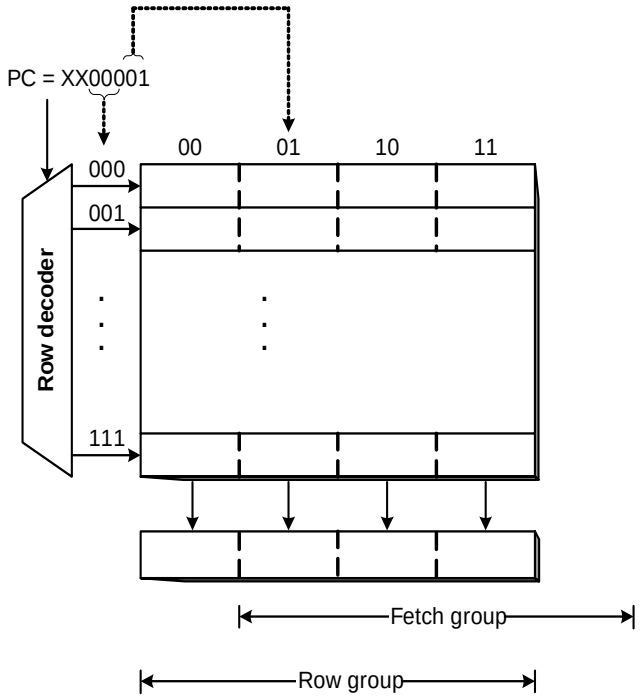
Slika 3.9 Organizacija I-keša: (a) jedna keš linija jednaka jednoj fizičkoj vrsti; (b) jedna keš linija odgovara dvema fizičkim vrstama

Napomena:: gde su:
Adrress- adresa; *Cache line* – keš linija; *Row decoder*- Dekoder vrsta; *Tag*- tag (adresni marker)

Primarni cilj stepena *Fetch* je da maksimizira propusnost kod pribavljanja instrukcija. Ostvarena propusnost koja se postiže od strane stepena *Fetch* ima uticaj na ukupnu propusnost *SSPM*-a jer propusnost svih narednih stepeni zavisi od nje i ne može da nadmaši propusnost stepena *Fetch*. Dva glavna faktora koja utiču na postizanje maksimalne propusnosti od *s* instrukcija po ciklusu su:

1. nepodešenost (neporavnjanost) *s* instrukcija koje se pribavljaju, a nazivaju se *fetch* grupa, u odnosu na organizaciju vrste I-keša (slučaj kada jedna *fetch* grupa pripada dvema vrstama)
2. egzistencija neke sekvence upravljanja kod pribavljanja instrukcija koje pribadaju *fetch* grupi (ako se pribavljaju instrukcije iz različitih vrsti tada je potrebno generisati veći broj adresa pristupa kao i upravljačke signale za čitanje)

Kod svakog mašinskog ciklusa *Fetch* stepen koristi PC da indeksira I-keš u cilju pribavljanja svih *s* instrukcija. Ako je cela *fetch* grupa memorisana u istoj vrsti keša, tada se svih *s* instrukcija pribavljaju odjedanput. Sa druge strane, ako *fetch* grupa prelazi granice vrste tada se svih *s* instrukcija ne mogu pribaviti u jednom ciklusu (usvajamo da se samo jednoj vrsti I-keša može pristupiti svakog ciklusa). To znači da se samo instrukcije prve vrste mogu pribaviti, dok ostale instrukcije zahtevaju drugi ciklus za pribavljanje. U tom slučaju propusnost pribavljanja efektivno se smanjuje na pola jer se zahteva dva ciklusa za pribavljanje *s* instrukcija. Ovo je posledica nepodešenosti *fetch* grupa, kod I-keša, u odnosu na granice vrste (vidi sliku 3.10). Ovakva nepodešenost redukuje efektivnu propusnost pribavljanja. U slučaju kada svaka keš linija odgovara fizičkoj vrsti, kako je to prikazano na slici 3.9 a) tada prelazak granice vrste takodje odgovara prelasku granice keš linije što ne uzrokuje dodatne probleme. Ako se *fetch* grupa prostire u dve keš linije tada može da se javi promašaj kod I-keša za slučaj kada se pristupa drugoj liniji, a da je pri tome prva linija rezidentna u kešu. Čak i u slučaju da su ove linije rezidentne u kešu fizički pristup većem broju keš linija u jednom ciklusu je problematičan.



Slika 3.10 Neporavnjanost *fetch* grupe u odnosu na granice vrste keš polja

Napomena:: *Row decoder*- Dekoder vrsta; *Row group*- grupa vrste; *Fetch group*- *Fetch* grupa

Postoje dva moguća rešenja koja se odnose na problem nepodešenosti. Prvo rešenje se zasniva na statičkoj tehnici koja se koristi u vreme kompilacije. Kompilatoru se dostavlja informacija o organizaciji I-keša, tj. njegovoj šemi indeksiranja i obimu vrste. Na osnovu ove informacije instrukcije se mogu na adekvatan način smestiti u memorijske lokacije i postići podešavanje *fetch* grupa u odnosu na fizičke vrste. Na primer, svaka instrukcija koja je cilj grananja se može smestiti u memoriju na lokaciju koja odgovara prvoj instrukciji u vrsti. Ovakav pristup povećava verovatnoću pribavljanja *s* instrukcija od početka vrste. Ovakve tehnike su implementirane kod velikog broja procesora i veoma su efikasne. Problem koji se javlja u ovom slučaju je taj što se objektni kôd prilagodjava pojedinoj I-keš organizaciji, pa može da se desi da on ne bude korektno poravnat za druge I-keš organizacije. Drugi problem koji se javlja da statički kôd zauzima veći adresni prostor, a to potencijalno predstavlja opasnost za povećanje promašaja kod obraćanja I-kešu.

Drugo rešenje koje se odnosi na nepodešenost bazira se na korišćenju usluge hardvera u toku izvršenja programa. Hardver poravnanja se može implementirati sa ciljem da obezbedi pribavljanje od *s* instrukcija svakog ciklusa čak i u slučaju kada *fetch* grupa predje granicu vrste (ali ne i granice keš linije). Ovakav hardver poravnanja je ugradjen kod procesora IBM RS/6000.

3.3.2. Dekodiranje instrukcija

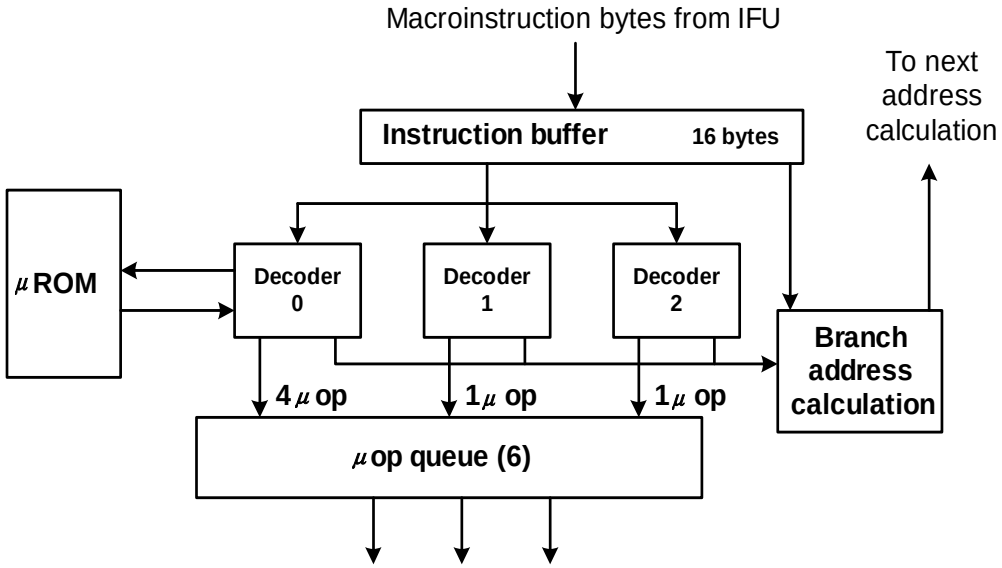
Dekodiranje instrukcije uključuje identifikaciju individualnih instrukcija, odredjivanje tipova instrukcija, i detekciju medju-instrukcijske zavisnosti izmedju grupa instrukcija koje su pribavljene, ali nisu još dispečovane (rasporedjene). Kompleksnost zadatka dekodiranja instrukcija jako zavisi od sledeća dva faktora: a) *ISA*, i b) od obima paralelnog protočnog sistema. RISC procesori imaju instrukcije fiksne dužine koje su jednostavnog formata, što ima za posledicu da proces dekodiranja bude pojednostavljen. To znači da nije potrebno činiti dodatne napore kao bi se odredio početak i kraj svake instrukcije. Relativno mali broj različitih instrukcionih formata i adresnih načina rada čini proces raspoznavanja tipova instrukcija veoma lakim. Jednostavnim dekodiranjem dela instrukcije koji je dodeljen op-kôdu, kao i ostala polja koja specificiraju operande čini proces dekodiranja veoma brzim.

Kod RISC *SPM*-ova, dekodiranje instrukcije je trivijalno. Najčešće stepen *Decode* se koristi za pristup registarskim operandima i uključuje aktivnost kojom se obavlja čitanje sadržaja registra *RF* polja. Ipak kod RISC paralelnog protočnog sistema koje simultano dekodiraju veći broj instrukcija, stepen *Decode* mora da identifikuje zavisnosti izmedju instrukcija i odredi nezavisne instrukcije koje se mogu paralelno dispečovati. Šta više da bi podržao efikasno pribavljanje instrukcija stepen *Decode* mora brzo da identifikuje promene toka upravljanja zbog *Branch* instrukcije i instrukcije koje se dekodiraju sa ciljem da obezbedi brzi povratni odziv prema stepenu *Fetch*. Ova dva zadatka, u saglasnosti sa istovremenim pristupom većem broju operanada, mogu da učine logiku stepena *Decode* kod paralelnih protočnih stepena veoma složenom. Da bi se odredile registarske zavisnosti izmedju instrukcija neophodno je ugraditi veći broj komparatora. Registarska polja moraju da budu tipa multiport i da budu u stanju da podrže veći broj simultanih pristupa. Veći broj magistrala je takodje potreban za rutiranje operanada ka odgovarajućim odredišnim baferima. Može da se desi da stepen *Decode* postane kritični stepen *SSPM*-a.

Kod CISC protočnih paralelnih sistema zadatak dekodiranja instrukcija može biti veoma kompleksan, a često i da zahteva veći broj protočnih stepena. Kod ovakvih paralelnih protočnh sistema, identifikacija individualnih instrukcija i njihovih tipova nije više trivijalna. Kod oba, Intel Pentium i AMD K5, se koriste dva protočna stepena za dekodiranje instrukcija IA32. Kod Intel PentiumPro, koga karakteriše dublja protočnost, neophodna su ukupno pet mašinska ciklusa za pristup I-kešu i dekodiranje instrukcija IA32. Korišćenje instrukcija promenljive dužine dovodi do nepoželjne serijelizacije u procesu dekodiranja instrukcija. To znači da se vodeća (prvo pribavljena) instrukcija mora dekodirati, i odredi njena dužina, pre nego što počne identifikacija naredne instrukcije. Saglasno tome simultano paralelno dekodiranje većeg broja instrukcija postaje veliki projektantski izazov. U najgorem slučaju mora se usvojiti da nova instrukcija može da počne bilo gde u okviru *fetch* grupe, a

veći broj dekodera može se koristiti za simultano i "spekulativno" dekodiranje instrukcija, počev od granice na nivou bajta. Ovo je ekstremno složen zadatak i može da bude veoma neefikasan.

Postoji dodatna poteškoća kod realizacije dekodera instrukcija kod CISC paralelnih protočnih sistema. Dekoder mora da prevede instrukcije u interne operacije niskog nivoa koje se mogu direktno izvršavati od strane hardvera. Ove interne operacije podsećaju na RISC instrukcije i mogu se posmatrati kao vertikalne mikroinstrukcije. Kod AMD K5 mikroprocesora ove operacije se nazivaju *RISC operations* ili *ROP* . Kod Intela ove interne operacije se identifikuju kao mikrooperacije ili μops. Svaka IA32 instrukcija se prevodi u jednu ili veći broj *ROP*-ova ili μops-ova. Kako to Intel-ovi projektanti naglašavaju u proseku svaka IA32 instrukcija se prevodi u 1.5 do 2.0 μops-ova. Kod ovih paralelnih CISC protočnih mašina izmedju stepena *Decode* i stepena *Complete* sve instrukcije koje se unose u mašinu dekomponuju se na ove interne RISC operacije.



Slika 3.11 *Fetch/Decode* jedinica kod superskalarnog protočnog sistema Intel P6

Napomena: *Macroinstruction bytes from IFU*- bajtovi mikroinstrukcije od instrukcione *Fetch* jedinice; *Instruction buffer*- instrukcioni bafer; 16 *bytes*- 16 bajtova; *Decoder*-dekoder; *μROM*-mikroprogramski ROM; *μop queue*- red čekanja mikroprogramskih operacija; *Branch adrress calculation*- izračunavanje adrese grananja; *To next adrress calculation*- ka izračunavanju naredne adrese

Dekoder instrukcija procesora Intel Pentium Pro predstavlja jedan ilustrativni primer realizacije dekodiranja instrukcija kod CISC paralelnih mašina. Dijagram *Fech/Decode* jedinice P6 procesora je prikazan na slici 3.11. U svakom mašinskom ciklusu I-keš može da preda redu čekanja instrukcije do 16 poravnjanih bajtova. Tri paralelna dekodera simultano dekodiraju bajtove instrukcije iz reda čekanja instrukcija (*instruction queue*). Prvi dekoder koji se nalazi na početku reda čekanja sposoban je da dekodira sve IA32 instrukcije, dok druga dva dekodera imaju ograničenije mogućnosti i mogu da dekodiraju jednostavne IA32 instrukcije kao što su instrukcije tipa *registar-u-registar*.

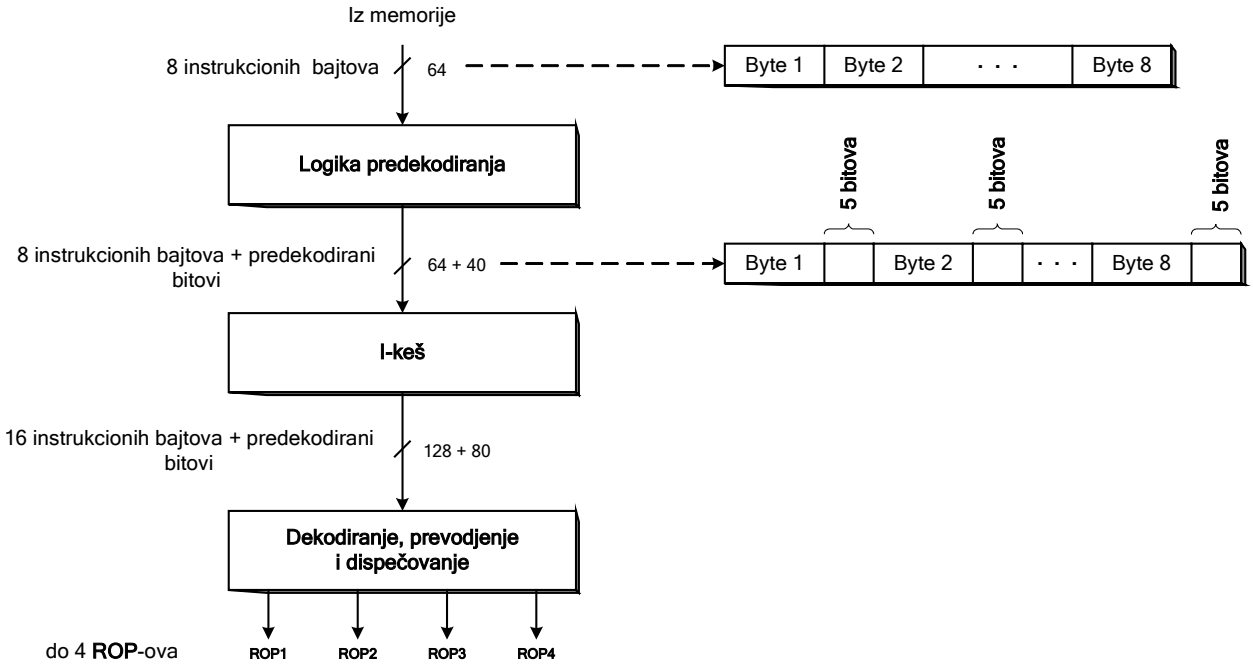
Dekoderi prevode IA32 instrukcije iz dvo-adresnog formata u interne tro-adresne μops formate, pri čemu μops koriste *Load/Store* model. Svaka IA32 instrukcija koja koristi složene adresne načine rada se prevodi u veći broj μops-ova. Prvi (generalizovani) dekoder može da generiše do 4 μops po ciklusu kao odziv na proces dekodiranja IA32 instrukcija. Svaki od ostala dva (sa izvedenim restrikcijama) dekodera može da generiše samo jednu μops po ciklusu kao odziv na dekodirane

jednostavne IA32 instrukcije. U svakom mašinskom ciklusu najmanje jedna IA32 instrukcija se dekodira od strane generalizovanog dekodera, što dovodi do generisanja jedne ili veći broj μops-ova. Cilj je da se postigne bolji rezultat i da se iskoriste mogućnosti ostala dva dekodera čime se broj μops poveća. U idealnom, slučaju tri paralelna dekodera mogu da generišu ukupno 6 μops u jednom mašinskom ciklusu. Za one kompeksne IA32 instrukcije koje zahtevaju više od 4 μops-ova za prevodjenje, u trenutku kada naidju na početak (*front*) reda čekanja, generalizovani dekoder poziva μops sekvencer da bi emitovao mikrokôd, što u suštini predstavlja jednostavnu unapred programiranu sekvencu standardnih μops. Za ove μops potrebna su dva ili više mašinska ciklusa. Sve μops generisane od strane triju paralelnih dekodera se pune u bafer preuredjenja (*reorder-buffer*, *ROB*), koji inače za 10 ulaza za čuvanje do 40 μops, koje čekaju na **dispečovanje** (alternativno nazvano rasporedjivanje ili dodeljivanje) ka funkcionalnim jedinicama.

Kod velikog broja *SSPM*-a, posebno kod onih koje implementiraju CISC paralelni protočni sistem šireg obima, hardver za dekodiranje instrukcija može da bude izuzetno složen i da se zahteva njegova parcijalizacija na veći broj protočnih sistema. Kada se broj dekoderskih stepena poveća, cena koja se plaća zbog instrukcije *Branch*, izražena u broju mašinskih ciklusa, se takodje povećava. Zbog toga nije poželjno da se samo poveća dubina *Decode* dela kod paralelne protočne mašine. Da bi se izašlo na kraj sa složenošću ovog problema, često se predlaže i implementira tehnika nazvana predekodiranje (*predecoding*).

Predodiranje prebacuje deo zadatka dekodiranja na drugu stranu, tj. na ulaznu stranu I-keša. Kada se javi I-keš promašaj dobavlja se nova linija iz memorije, a instrukcije u toj keš liniji parcijalno se dekodiraju od strane hardvera za dekodiranje koji se nalazi izmedju memorije i I-keša. Instrukcije kao i neke dodatno dekodirane informacije se zatim smeštaju u I-keš. Dekodirana informacija, u formi predekodiranih bitova, pojednostavljuje zadatak dekodiranja instrukcija kada se instrukcije pribavljaju iz I-keša. Saglasno tome deo procesa dekodiranja se obavlja samo jedanput kada se instrukcije pune u I-keš, umesto svaki put kada se instrukcije pribavljaju iz I-keš. Sa ovakvim hardverom za dekodiranje koji je prebačen na ulaznu stranu I-keša, kompeksnost dekodiranja istrukcija kod paralelnog protočnog sistema se pojednostavljuje.

AMD K5 je tipičan predstavnik CISC *SSPM*-ova koji koriste agresivno predekodiranje IA32 instrukcije kada se one pribavljaju iz memorije, ali pre nego što se smeste u I-keš. U jednoj transakciji po magistrali pribavljaju se iz memorije 8 instrukcionih bajtova. Ovi bajtovi se predekodiraju i dodatnih 5 bitova se generiše od strane predekodera za svaki instrukcioni bajt. Ova 5 predekodirana bita sadrže informaciju o lokaciji početka i kraja IA32 instrukcije, broju μops (ili *ROP*-ova) koji je potreban da se prevede ta IA32 instrukcija, kao i lokaciju op-kôd-ova i prefiksa. Ovi dodatni predekodirani bitovi se smeštaju u I-keš zajedno sa originalnim instrukcijskim bajtovima. Saglasno tome obim originalne I-keš linije od 128 bitova (16 bajtova) se povećava za dodatnih 80 bitova (vidi sliku 3.12).



Slika 3.12 Mehanizam predekodiranja kod AMD K5

Kod svakog I-keš pristupa zajedno sa 80 predekodiranih bitova pribavljaju se i 16 instrukcionih bajtova. Predekodirani bitovi u značajno meri pojednostavljaju dekodiranje instrukcije i obezbedjuju simultano dekodiranje većeg broja IA 32 instrukcije pomoću 4 identična dekodera/translatora koji mogu da generišu do 4 μ ops svakog ciklusa.

Postoje dve forme prekoračenja (*overhead*-dodatno režijsko vreme) koja prate predekodiranje. Cena koja se plaća kod I-keš promašaja može da se poveća zbog potrebe predekodiranja instrukcijskih bajtova koji se pribavljaju iz memorije. Ovo i nije neki ozbiljan problem ako je stopa promašaja mala. Drugo prekoračenje se odnosi na pamćenje predekodiranih bitova u I-kešu, a saglasno tome povećanju obima I-keša. Kod K5 obim I-keša se povećava za oko 50%. Na osnovu ovoga jasno je da mora da postoji kompromis izmedju agresivnosti predekodiranja i povećanja obima I-keša.

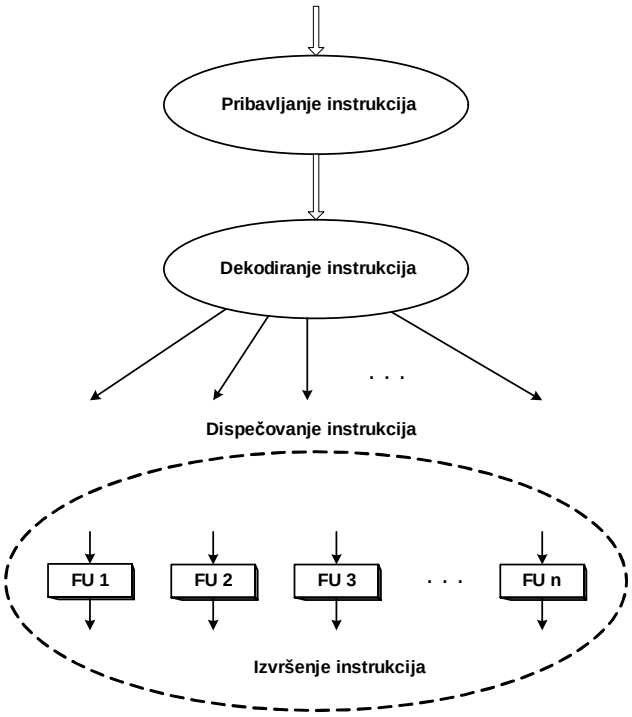
Predekodiranje nije samo ograničeno na otklanjanju sekvencijalnog uskog grla kod paralelnog dekodiranja većeg broja CISC instrukcija od strane CISC paralelnih protočnih sistema. Ono se takodje može koristiti za podršku rada RISC paralelnih protočnih sistema. RISC instrukcije se mogu predekodirati kada se one napune u I-keš. Predekodirani bitovi se mogu koristiti za identifikaciju promene toka upravljanja kod *Branch* instrukcija u okviru *fetch* grupe i za eksplicitnu identifikaciju podgrupa nezavisnih instrukcija u okviru *fetch* grupe. Na primer, PowerPC 620 koristi 7 predekodiranih bitova za svaku reč instrukcije u I-kešu. UltraSPARC, MIPS R10000 i HP PA-8000 takodje koriste 5 ili 4 predekodiranih bitova za svaku instrukciju.

Kako *SSPM* postaje širi, a broj instrukcija koje se mogu simultano dekodirati povećava zadatak, dekodiranja instrukcije predstavlja sve više usko grlo pa zbog toga agresivnije korišćenje predekodiranja postaje neminovnost. Predekoder parcijalno dekodira instrukcije, i efikasno transformiše originalne nedekodirane instrukcije u format koji čini zadatak konačnog dekodiranja lakšim. U suštini predekoder se može posmatrati kao translator instrukcija koje se pribavljaju iz memorije u neki drugi različiti oblik instrukcija koje se pune u I-keš. Da bi proširili ove poglede mogućnost poboljšanja predekodera koji će u toku izvršenja programa obaviti transliranje objektnog kôda izmedju *ISA*-ova može da bude veoma interesantan izazov za projektante.

3.3.3. Dispečovanje instrukcija

Dispečovanje intrukcija je neophodno da se obavi kod *SSPM*-ova. Kod *SPM*-a sve instrukcije nezavisno od tipa prolaze kroz jedinstveni protočni sistem. *SSPM*-ovi su raznorodni sa aspekta protočnosti po tome što koriste veći broj heterogenih funkcionalnih jedinica u njihovom *Execution* stepenu. Različite funkcionalne jedinice izvršavaju različiti tipove instrukcija. Nakon što je tip instrukcije identifikovan od strane stepena *Decode*, ona mora da se uputi ka odgovarajućoj funkcionalanoj jedinici radi izvršenja. Ovaj zadatak se naziva dipsečovanje instrukcija.

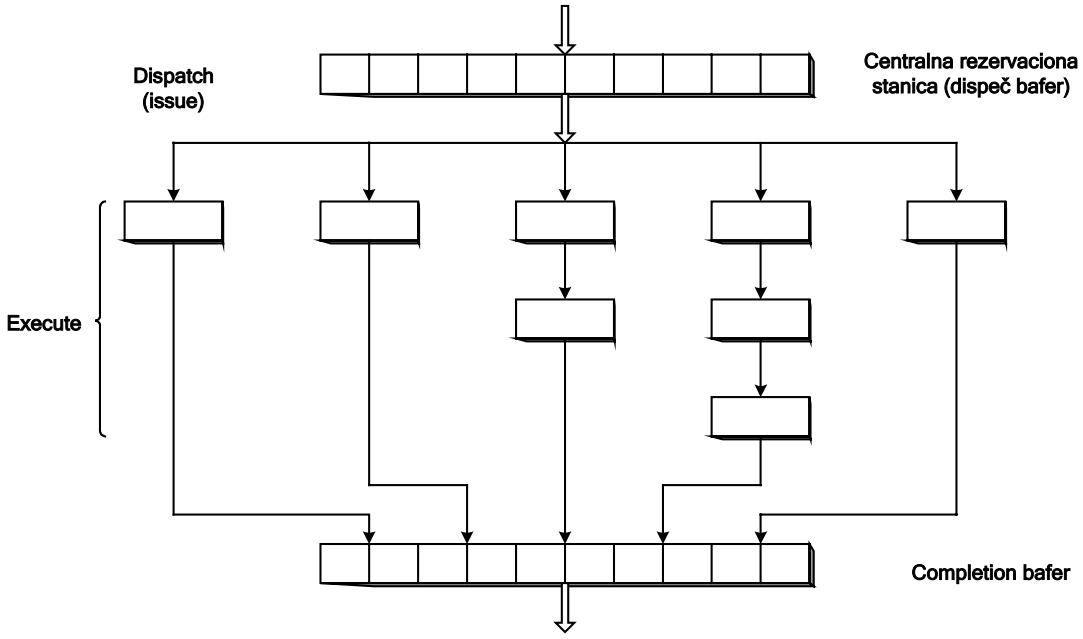
Mada su *SSPM*-ovi paralelne protočne mašine, zadaci koji se odnose na pribavljanje instrukcije kao i njihovo dekodiranje izvode se na jedan centralizovan način; tj. progres svih instrukcija se nadgleda od strane jednog kontrolera. Mada se veći broj instrukcija pribavlja u jednom ciklusu, sve instrukcije moraju da se pribave iz istog I-keša. To znači da se svim instrukcijama iz *fetch* grupe koje se nalaze u I-kešu pristupa istovremeno, a zatim se sve one smeštaju u isti bafer. Dekodiranje instrukcija se izvodi na centralizovani način jer u slučaju CISC instrukcija, svi bajtovi u *fetch* grupi moraju biti dekodirani kolektivno od strane centralizovanog dekodera kako bi se identifikovale individualne instrukcije. Čak i kod RISC instrukcija, dekoder mora da identifikuje zavisnosti izmedju instrukcija, pa je zbog toga neophodno centralizovano dekodiranje instrukcija. Sa druge strane kod raznorodnih protočnih sistema sve funkcionalne jedinice mogu da rade nezavisno na jedan distribuirani način u toku izvršenja različitih tipova instrukcija nakon što su zavisnosti izmedju instrukcija razrešene. Saglasno tome, prelazak sa procesa dekodiranja instrukcija na proces izvršenja instrukcija uključuje promenu sa centralizovanog procesiranja instrukcija na distribuirano procesiranje instrukcija. Ova promena se izvodi od strane stepena za dispečovanje instrukcije koji je, kao što se vidi na slici 3.13, sastavni gradivni blok *SSPM*-a.



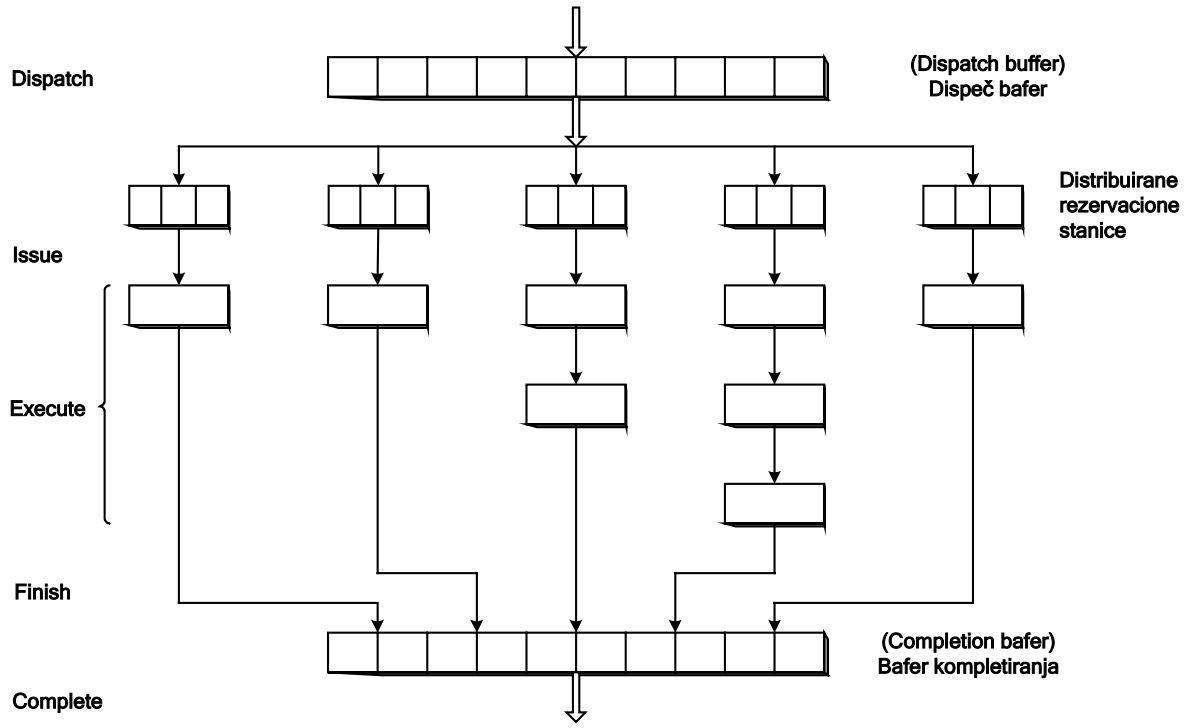
Slika 3.13 Potreba za dispečovanje kod superskalarnog protočnog sistema

Jedan drugi mehanizam koji je neophodan da postoji izmedju dekodiranja instrukcija i izvršenja instrukcija se odnosi na privremeno baferovanje instrukcija. Pre izvršenja, instrukcija mora da ima spremne sve svoje operande. U toku dekodiranja registarski operandi se pribavljaju iz *RF* polja. Kod *SSPM*-a moguće je da neki od ovih operanada nisu još spremni jer prethodne instrukcije koje ažuriraju ove registre nisu završile svoje izvršenje. Kada dodje do ovakve situacije, jedno od rešenja je da se zaustavi *Decode* stepen sve dok svi registarski operandi ne budu spremni. Ovo rešenje ozbiljno ograničava propusnost kod dekodiranja i kao takvo nije poželjno. Bolje rešenje je da se pribave oni registarski operandi koji su spremni, a da se ostale instrukcije koje će čekati na svoje registarske operande, koji još nisu spremni, smeste u posebne bafere. Kada su svi registarski operandi spremni, te instrukcije napuštaju bafer i one se predaju radi izvršenja funkcionalnim jedinicama. Koristeći terminologiju *Tomasulo*-ovog algoritma koji se koristio kod IBM 360/91, nazvaćemo ovaj privremeni bafer za instrukcije imenom rezervaciona stanica (*reservation station*). Korišćenjem rezervacine stanice vrši se razdvajanje procesa dekodiranja od procesa izvršenja instrukcija, i obezbedjuje baferu da formira strukturu tipa magacin izmedju stepena *Decode* i *Execute*. Na ovaj način se ostvaruje ublažavanje vremenskih varijacija u iznosu propusnosti izmedju stepena *Decode* i *Execute*. Drugim rečima eliminišu se zastoji u stepenu *Decode* i nepotrebna blokiranja u stepenu *Execute*.

U zavisnosti od toga gde je locirana rezervaciona stanica, relativno u odnosu na dispečovanje instrukcija, postoje dva tipa implementacije rezervacione stanice. Ako se koristi jedinstveni bafer na izvornoj strani dispečovanja, tada kažemo da postoji centralizovana rezervaciona stanica. Kada se koristi veći broj bafera na izvornoj strani dispečovanja, tada kažemo da postoje distribuirane rezervacione stanice. Na slici 3.14 i 3.15 prikazana su oba načina implementacije rezervacionih stanica.



Slika 3.14 Centralizovana rezervaciona stanica



Slika 3.15 Distribuirane rezervacione stanice

Intel Pentium Pro ima implementiranu centralizovanu rezervacionu stanicu. Kod ove implementacije jedna rezervaciona stanica sa većim brojem ulaza predaje podatke svim funkcionalnim jedinicama. Instrukcije se direktno rasposedjuju od ove centralizovane rezervacione stanice ka svim funkcionalnim jedinicama da bi počele sa izvršnjem. Sa druge strane Power PC 620 koristi distribuirane rezervacione stanice. Kod ove implementacije, svaka funkcionalna jedinica ima sopstvenu rezervacionu stanicu koja je povezana na ulaznoj strani te jedinice. Instrukcije se rasporedjuju individualnim rezervacionim stanicama na osnovu tipa instrukcija. Ove instrukcije ostaju u rezervacionim stanicama sve dok ne postanu spremne za iniciranje njihovog izvršenja u funkcionalnoj jedinici. Naravno ova dva tipa implementacije rezervacionih stanica prestavljaju dve ekstremne alternative. Hibridna rešenja koja se nalaze izmedju ova dva rešenja su moguća. Na primer, MIPS R10000 koristi jednu takvu hibridnu implementaciju. Ovakve hibridne implementacije nazivaju se *cluster* rezervacine stanice. Kod *cluster* rezervacionih stanica, instrukcije se rasporedjuju većem broju rezervacionih stanica, pri čemu svaka rezervaciona stanica može da predaje i da bude deljiva za veći broj funkcionalnih jedinica. Obično rezervacione stanice i funkcionalne jedinice kažemo da se *cluster*-uju na osnovu tipa instrukcija ili tipa podataka.

Kod projektovanja rezervacionih stanica neophodno je učiniti odredjeni broj kompromisa. Centralizovana rezervaciona stanica omogućava svim tipovima instrukcija da dele istu rezervacionu stanicu i verovatno postižu najbolju ukupnu iskorišćenost ulaza u odnosu na sve rezervacione stanice. Ipak centralizovana implementacija rezultira najvećom hardverskom kompleksnošću. Ona zahteva centralizovano upravljanje i multiportni bafer koji obezbedjuje veći broj konkurentnih pristupa. Kod distribuiranih rezervacionih stanica potrebno je ugradjivati jednoportne bafere, sa relativno malim brojem ulaza. Ipak svaki ulaz rezervacione stanice u kome se nalazi podatak koji čeka na izvršenje ne može se usmeriti za izvršenje ka drugim funkcionalnim jedinicama. To znači da je ukupna iskorišćenost svih ulaza rezervacionih stanica mala. Takodje može da se desi da se jedna rezervaciona stanica zasiti, u slučaju kada svi ulazi budi zauzeti, što dovodi do zastoja u rasporedjivanju instrukcija.

Imajući u vidu različite alternative koje se odnose na implementaciju rezervacionih stanica pre nego što detaljnije shvatimo suštinu rada, neophodno je da razjasnimo korišćenje odredjenih termina.

U ovom tekstu termin **rasporedjivanje (*dispatching*)** ukazuje na to kako se vrši pridruživanje različitih tipova instrukcija, radi izvršenja, odgovarajućim tipovima funkcionalnih jedinica, nakon što je obavljen proces dekodiranja.

Sa druge strane termin **iniciranje izvršenja (*issuing*)** uvek se odnosi na iniciranje izvršenja u funkcionalnim jedinicama. Kod distribuirane rezervacione stanice, ova dva dogadja egzistiraju izdvojeno. Instrukcije se prvo rasporedjuju iz centralizovanog bafera za *decode/dispatch* ka individualnim stanicama, a kada su svi njihovi operandi dostupni, one se zatim iniciraju radi izvršenja u individualnim funkcionalnim jedinicama. Kod centralizovane rezervacione stanice, do rasporedjivanja instrukcija iz centralizovane rezervacione stanice ne dolazi sve dok operandi ne budu spremni. To znači da se sve instrukcije, nezavisno od tipa, čuvaju u centralizovanoj rezervacionoj stanici dok one ne budu spremne za izvršenje, pa se nakon tog trenutka instrukcije direktno rasporedjuju u individualne funkcionalne jedinice sa ciljem da bi počele sa izvršenjem. Shodno tome, kod mašina sa centralizovanom rezervacionom stanicom, pridruživanje instrukcija individualnim funkcionalnim jedinicama javlja se u istom trenutku kao i početak njihovog izvršenja. Zbog toga kod centralizovane rezervacione stanice, rasporedjivanje (*dispatching*) instrukcije i iniciranje (*issuing*) izvršenja javlja se istovremeno, tako da ova dva termina imaju isto značenje. Ovo je prikazano na slici 3.14.

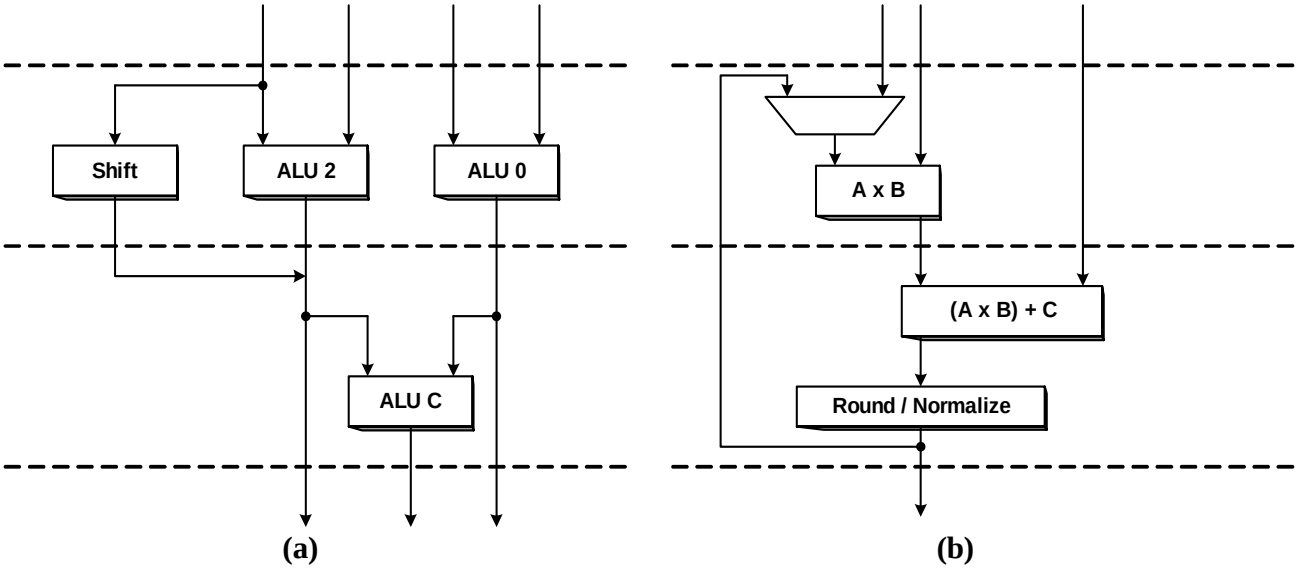
3.3.4. Izvršenje instrukcija

Stepen za izvršenje instrukcija predstavlja ključni deo superskalarne mašine. Tekući trend kod *SSPM*-a je usmeren ka ostvarivanju većeg paralelizma i veće raznovrsnosti u protočnoj obradi. Ovo znači da postoji veći broj funkcionalnih jedinica i da su te funkcionalne jedinice specijalizovane. Ako ove jedinice specijaliziramo radi obavljanja specifičnih tipova instrukcija, tada će one biti efikasnije sa aspekta performansi. Prvi *SPM*-ovi imali su samo jednu funkcionalnu jedinicu . Svi tipovi instrukcija (isključujući *floating-point* instrukcije koje su bile izvršavane od strane posebnog *FP* koprocesorskog čipa) izvršavale su se od strane iste funkcionalne jedinice. Tako na primer, kod protočnog sistema TYP, ovu funkcionalnu jedinicu su činila dva protočna stepena nazvana ALU i MEM. Najveći broj *SSPM*-ova prve generacije su protočne mašine kod kojih ima ugradjeno dve raznorodne funkcionalne jedinice, jedna koja izvršava *integer* instrukcije, a druga *floating-point* instrukcije. To znači da su prvi *SSPM*-ovi imali integrisano *FP* izvršenje u okviru iste protočne obrade instrukcija, a nisu koristile izdvojenu koprocesorsku jedinicu.

Današnji *SSPM*-ovi koriste veći broj *integer* jedinica, a neke i imaju veći broj *floting-point* jedinica. One u suštini predstavljaju dva osnovna tipa funkcionalnih jedinica. Neke od ovih jedinica su postale veoma moćne i u stanju su da izvrše više od jedne operacije pribavljajući po dva izvorna operanda u svakom ciklusu. Na slici 3.16 a) prikazana je *integer* izvršna jedinica procesora TI Super SPARC koja ima kaskadnu ALU konfiguraciju. Tri ALU jedinice sastavni su deo jedne dvostepene protočne jedinice, i po dve *integer* operacije se mogu u jednom ciklusu inicirati radi izvršenja u ovoj jedinici. Ako su one nezavisne, tada se obe operacije izvršavaju u prvom stepenu koristeći ALU 0 i ALU 2. Ako druga operacija zavisi od prve, tada se prva izvršava u ALU 2 koji pripada prvom stepenu, dok se druga izvršava u ALU C koji pripada drugom stepenu. Implementacija ovakve funkcionalne jedinice omogućava simultano iniciranje radi izvršenja dve instrukcije.

FP jedinica kod IBM RS/6000 je implementirana kao dvostepena protočna *MAF*(*mulply–add-fused*) jedinica koja ima tri ulaza (A,B,C) i obavlja $(\boldsymbol{A}\times\boldsymbol{B})+\boldsymbol{C}$. Ovo je prikazano na slici 3.19 b) . Opravdanost ugradnje *MAF* jedinice pre svega treba tražiti u veoma čestom korišćenju operacije FP množenje koja se izvodi za izvršenje skalarne ili *dot-product* operacije $\boldsymbol{D}=(\boldsymbol{A}\times\boldsymbol{B})+\boldsymbol{C}$. Ako je kompajler u stanju da objedini veći broj parova instrukcija tipa množenje sabiranje u jedinstvene *MAF* instrukcije, a MAF jedinica može da podrži iniciranje izvršenja od jedne *MAF* instrukcije svakog

taktnog ciklusa, tada se korišćenjem jedne MAF jedinice može ostvariti efektivna propusnost od dve *FP* instrukcije po ciklusu. Normalne *FP* instrukcije množenja se u suštini izvršavaju od strane MAF jedinice kao $(A \times B)+0$, dok se *FP* instrukcija sabiranja obavlja od strane MAF jedinice kao $(A \times 1)+C$. Sobzirom da je MAF jedinica protočno organizovana ona može da postigne brzinu izvršenja od jedne *FP* instrukcije po ciklusu.



Slika 3.16 (a) *Integer* funkcionalna jedinica kod TI SuperSPARC; (b) *Floating-point* jedinica kod IBM RS/6000

Napomena:: *Shift*- pomeranje; *Round/Normalize*-zaokruživanje/normalizaciju

Pored izvršenja *integer ALU* instrukcija, integer jedinica se može koristiti za generisanje memorijskih adresa kao i izvršenje *Branch* i *Load/Store* instrukcija. Ipak kod novijih rešenja ugradjene su posebne jedinice za Branch i Load/Store. Branch jedinica je odgovorna za ažuriranje PC-a, dok je Load/Store jedinica direktno povezana na D-keš. Postoji druge specijalizovane jedinice koje se uglavnom koriste za podršku rada kod procesiranja u grafičkim i aplikacijama obrade slike. Na primer, kod Motorola 88110 postoji nameska funkcionalna jedinica za manipulisanje sa bitovima i dve funkcionalne jedinice za procesiranje *pixel*-a (osnovni element slike). Kod velikog broja aplikacija tipa signal procesiranje i medija uobičajeno se manipuliše sa podacima tipa bajt. Najčešće se 4 bajta pakovana u 32-bitnu reč, u cilju povećanja propusnosti, simultano procesiraju od strane specijalizovanih 32-bitnih funkcionalnih jedinica. Kod TriMedia VLIW procesora koji je namenjen za ovakve aplikacije, koriste se ove funkcionalne jedinice. Na primer, TriMedia 1 procesor može da izvrši *quadavg* instrukciju u jednom ciklusu. *Quadavg* instrukcija sumira četiri srednje vrednosti zaokružene na celobrojnu vrednost naviše, i veoma je korisna kod MPEG dekodiranja za dekompresiju komprimovanih video slika, a izvodi se na sledeći način:

$$quadavg = \frac{(a+e+1)}{2} + \frac{(b+f+1)}{2} + \frac{(c+g+1)}{2} + \frac{(d+h+1)}{2}$$

Osam promenljivih ukazuju na osam operanada od kojih su *a*, *b*, *c* i *d* memorisane kao jedna 32-bitna vrednost, dok su *e, f, g* i *h* memorisane kao druga 32-bitna vrednost. Funkcionalna jedinica prihvata kao ulaz ova dva 32-bitna operanda i generiše *quadavg* rezultat u jednom ciklusu. Ova jedno-

ciklusna operacija zamenjuje brojne instrukcije sabiranja i deljenja koje treba da se izvrše ako se individualno manipuliše sa operandima tipa bajt. Široko korišćenje multimedijalnih aplikacija dovelo je do još masovnijeg korišćenja ovakvih funkcionalnih jedinica.

Koji je najbolji izbor funkcionalnih jedinica kod *SSPM*-a ? predstavlja interesantno pitanje. Odgovor je da to zavisi od specifikacija. Ako koristimo statistike iz poglavlja 2 koje se odnose na tipične programe možemo uočiti da *ALU* instrukcije čine 40%, *Branch* 20%, a *Load/Store* 40%, što odgovara pravilu 4-2-4. To znači da na svake četiri AL jedinice, treba da imamo dve Branch jedinice i četiri Load/Store jedinice. Veliki broj današnjih *SSPM*-ova imaju po četiri i više funkcionalne jedinice tipa ALU (ubrajajući kako *integer* tako i *floating-point*). Najveći broj njih ima samo jednu Branch jedinicu, ali u stanju su da spekulativno izvršavaju i više od jedne *Branch* instrukcije. Ipak najveći broj od ovih procesora ima samo jednu Load/Store jedinicu, ali su neke u stanju da procesiraju dve *Load/Store* instrukcije svakog mašinskog ciklusa uz odredjena ograničenja. Po svemu izgleda postoji debalans koji se ogleda u tome da postoji relativno mali broj Load/Store jedinica. Razlog za ovo je da implementacija većeg broja Load/Store jedinica koje na paralelni način pristupaju istom D-kešu predstavlja težak projektantski zadatak. On iziskuje da D-keš bude multiportni. Multiportni memorijski moduli sadrže veoma složena kola koja mogu značajno da uspore brzinu rada memorije.

Kod velikog broja rešenja koristi se veći broj memorijskih banaka koje u odredjenom smislu na pravi način simuliraju rad miultiportnih memorija. Kod ovakvih rešenja memorija se deli na veći broj banaka. Svaka banka može da obavi operaciju tipa *read/write* u jednom mašinskom ciklusu. Ako se desi da se efektivne adrese dve *Load/Store* instrukcije nalaze u različitim bankama, tada obe instrukcije mogu istovremeno da se izvrše na dve različite banke. Ipak, ako postoji konflikt u pristupu bankama, tada obe instrukcije moraju da se serijalizuju. Za simulaciju multiportnih D-kešova obično se koriste miltibankovni D-keševi. Na primer, Intelov Pentium procesor koristi D-keš od 8 banaka koji simulira rad dvoportnog D-keša. Prava multiportna memorija garantuje simultane pristupe bez konflikta. Obično je potreban veći broj portova za čitanje u odnosu na upis. Veći broj portova za čitanje se može implementirati pomoću većeg broja kopija memorije. Svi memorijski upisi se predaju ka svim kopijama, pri čemu sve kopije imaju identični sadržaj. Svaka kopija može da obezbedi relativno mali broj portova za čitanje pri čemu je ukupan broj portova za čitanje jednak zbiru svih portova za čitanje od svih kopija. Na primer, memorija sa četiri porta za čitanje i dva porta za upis se može implementirati kao dve kopije jednostavnih memorijskih modula, pri čemu svaka ima samo dva porta za upis i dva porta za čitanje. Implementacija većeg broja (posebno više ode dve) Load/Store jedinica koje rade paralelno može da predstavlja izazov kod projektovanja *SSPM*-ova. Iznos paralelizma na nivou resursa koji se odnosi na izvršenje instrukcija odredjuje se kombinacijom prostornog i vremenskog paralelizma. Egzistencija većeg broja funkcionalnih jedinica predstavlja jednu formu prostornog paralelizma. Alternativno, paralelizam se može ostvariti korišćenjem protočne obrade u radu ovih funkcionalnih jedinica, što predstavlja odredjeni oblik vremenskog paralelizma. Tako na primer, umesto da implementiramo dual-portni D-keš, kod tekućih rešenja D-keš je uprotočen u dva protočna stepena tako da se po dve *Load/Store* instrukcije mogu opsluživati od strane D-keša. Tekuće, postoji opšti trend ka implementaciji dublje protočnosti sve sa ciljem da se redukuje vreme ciklusa i poveća taktna frekvencija rada. Ovaj prostorni paralelizam takodje zahteva ugradnju na silicijumu složenijeg hardvera. Vremenski paralelizam efikasnije koristi hardver, ali dovodi do povećanja latencije kod procesiranja instrukcija i do pojave zastoja u protočnom radu mašine zbog zavisnosti izmedju instrukcija.

Kod realnih *SSPM*-ova, uočavamo da ukupan broj funkcionalnih jedinica nadmašuje stvarni obim paralelne protočne mašine. Obično širina *SSPM*-a odredjena je brojem instrukcija koje se mogu pribaviti, dekodirati, ili kompletirati u svakom mašinskom ciklusu. Ipak, zbog dinamike u varijaciji *mix*-a instrukcija kao i rezultantne neuniformne distribucije *mix*-a instrukcija u toku izvršenja programa na pricipu *ciklus-po-ciklus*, postoji potencijalno dinamička neuparenost instrukcijsko *mix*-a sa jedne strane i *mix*-a funkcionalnih jedinica sa druge strane. Prvi *mix* varira u vremenu, dok drugi ostaje fiksan. Zbog specijalizovanosti i heterogenosti funkcionalnih jedinica ukupan broj funkcionalnih

jedinica mora da premaši obim *SSPM*-a kako bi se izbeglo da *EX* deo protočnog sistema postane usko grlo zbog izrazito velike strukturne zavisnosti izmedju instrukcija koje se pre svega odnose na nedostupnosti korišćenja odgovarajućih tipova funkcionalnih jedinica. Neki od agresivnijih kompajlera teže da ublaže dinamičku varijaciju instrukcionog *mix*-a i da obezbede bolju uparanost sa *mix*-om funkcionalnih jedinica. Naravno različiti aplikacioni programi mogu da imaju različite tipove *mix*-ova instrukcija.

Sa većim brojem funkcionalnih jedinica imamo dodatnu hardversku kompleksnost u odnosu na same *FU*. Rezultati sa izlaza *FU* trebaju da se premoste (proslede) ka ulazima drugih *FU*. Za ovo je potreban veći broj magistrala, a potencijalno i složenija logika za upravljanje arbitražom na magistrali. Obično potpuna *crosbar* sprežna mreža je suviše skupa i ne uvak u potpunosti opravdana. Mehanizam za rutiranje ili usmeravanje operanada izmedju *FU* unosi dodatni oblik strukturne zavisnosti. Sprežni mehanizam takodje povećava latenciju *EX* stepena protočnog sistema. Sa ciljem da se podrži prosledjivanje ili premošćavanje podataka rezervacione stanice mora da nadgledaju protok na magistrali sa ciljem da ostvare *tag* uparivanja, što ukazuje na dostupnost neophodnih operanada, a nakon toga lečovanje operanada kada oni postanu dostupni na magistrali. Potencijalno kompleksnost *EX* stepena se povećava sa n^2 gde je n - ukupan broj funkcionalnih jedinica.

3.3.5. Kompletiranje i izvlačenje instrukcija

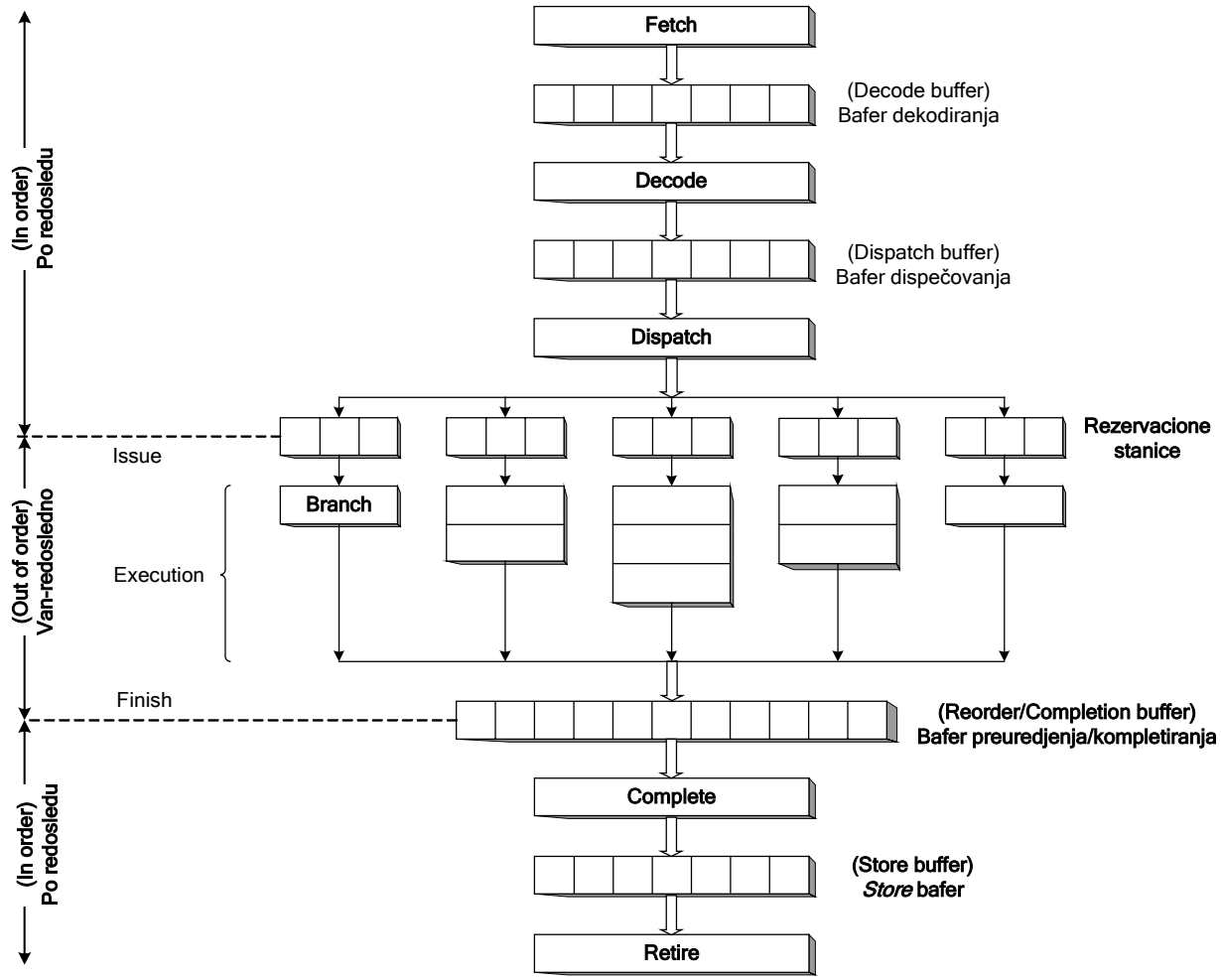
Funkciju smatramo kompletiranom (*completed*) kada ona završi sa izvršenjem i ažurira stanje mašine. Kada instrukcija završi sa svojim izvršenjem ona napušta funkcionalnu jedinicu i ulazi u bafer kompletiranja. Trenutak kada instrukcija napušta bafer kompletiranja kažemo da je postala kompletirana. Kada instrukcija završi sa izvršenjem, njen rezultat se može naći u nearhitekturnim baferima. Ali kada se ona kompletira njen rezultat se upisuje u arhitekturni registar. Kod instrukcija koje ažuriraju memorijske lokacije postoji vremenski period izmedju trenutka kada su one arhtekturno kompletirane i trenutka kada se memorijske lokacije ažuriraju. Na primer, instrukcija *Store* može biti arhitekturno kompletirana kada ona napusti bafer kompletiranja i udje u bafer memorisanja sa ciljem da sačeka za dostupnost na ciklus magistrale kako bi bila upisana u D-kešu. Instrukcija *Store* se smatra izvučena (*retired*) kada ona napusti bafer memorisanja i ažurira D-keš. Shodno tome u daljem tekstu pojam kompletiranja instrukcija (*instruction completion*) uključuje ažuriranje stanje mašine, dok pojam izvlačenja instrukcije (*instruction retiring*) podrazumeva ažuriranje stanja memorije. Kod instrukcija koje ne ažuriraju memoriju, izvlačenje i kompletiranje se dešavaju istovremeno. Tako na primer, kod mašine sa distribuiranom rezervacionom stanicom, instrukcija može da prodje kroz sledeće faze: *fetch, decode, dispatch, issue, execute, finish, complete,* i *retire*. A pri ovome faze *issue* i *finish* se odnose na početak *execute* i završetak *execute*, respektivno. Neki od proizvodjača *SSPM*-ova koriste ove termine na nešto različit način. Najčešće *dispatch* i *issue* se koriste sa istim značenjem, a slično i *complete* i *retire*. Ponekad *complete* se koristi da označi kraj *execute* dok *retire* se koristi da ukaže na ažuriranje arhitekturnog stanja mašine. U suštini ne postoji standardizovani način korišćenja ovih termina.

U toku izvršenja programa, mogu se javiti prekidi i izuzeci koji prekidaju izvršenje toka programa. *SSPM*-ovi koriste dinamičku protočnost koja olakšava *van-redosledno* izvršenje pa zbog toga moraju biti u stanju da uspešno izadju na kraj sa prekidima u izvršenju programa. Prekidi (*interrupts*) se obično iniciraju od strane spoljnjeg okruženja kakvi su ulazno/izlazni uredjaji ili operativni sistem. Ovi dogadjaji se javljaju asinhrono u odnosu na tekuće programsko izvršenje. Kada se javi prekid, izvršenje programa mora da se suspenduje kako bi se dozvolilo operativnom sistemu da opsluži prekid. Jedan od načina da se uradi ovo sastoji se u stopiranju pribavljanja novih instrukcija i dozvoljavanju instrukcijama koje se već nalaze u protočnom sistemu da završe sa izvršenjem, pri čemu tekuće stanje mašine mora da se zapamti. Nakon što operativni sistem opsluži prekid, obnavlja se zapamćeno stanje mašine, a prekinuti program nastavlja sa izvršenjem.

Izuzeci (*exceptions*) su dogadjaji iniciranji izvršenjem instrukcije uprogramu. Instrukcija može da inicira izuzetak kod izvršenja aritmetičkih operacija, kakve su deljenje sa nulom, i podbačaj (*underflow*) ili premašaj (*overflow*) kod manipulisanja sa brojevima u pokretnom zarezu. Kada se ovakav izuzetak javi, rezultati izračunavanja postaju nevalidni, pa operativni sistem mora da interveniše kako bi markirao ovakve izuzetke. Izuzeci se mogu takodje javiti zbog pojave greške kod straničenja kod stranično organizovanog virtuelnog memorijskog sistema. Ovi izuzeci se javljaju u toku obraćanja memoriji. Kada se javi ovaj izuzetak novu stranicu treba dobaviti iz sekundarne memorije, a to obično zahteva nekoliko hiljada mašinskih ciklusa. Saglasno tome, izvršenje programa koje je uzrokovalo grešku straničenja se obično suspenduje, a izvršenje novog programa u multiprogramskom režimu rada se inicira. Nakon što je greška u straničenju opslužena originalni program nastavlja sa izvršenjem.

Važno je napomenuti da arhitekturno stanje mašine, koje je važeće u trenutku kada se očekivana instrukcija izvršava, bude zapamćeno tako da program može nastaviti sa izvršenjem nakon što se izuzetak opsluži. Za mašine koje su u stanju da podrže ovu suspenziju i nastavak izvršenja programa na nivou granularnosti induividualne instrukcije kažemo da imaju precizni izuzetak (*precise exception*). Precizni izuzetak podrazumeva da se markira stanje mašine pre izvršenja instrukcije za obradu izuzetka, a nakon toga da se nastavi sa izvršenjem, obnavljanjem markiranog stanja, i restartovanjem izvršenja od očekivane instrukcije. Sa ciljem da se podrži rad preciznog izuzetka, *SSPM*-ovi moraju da održavaju arhitekturno stanje i da evoluiraju ovo mašinsko stanje na način kao da se programske instrukcije izvršavaju u vremenu u originalnom programskom redosledu (a ne *van-redosledno*). Razlog za ovo je sledeći: kada se javi izuzetak, stanje mašine u tom trenutku mora da odražava uslov koji se odnosi na to da sve instrukcije koje prethode pojavi instrukcije koja je generisala izuzetak moraju da se kompletiraju, a sve ostale instrukcije koje slede nakon instrukcije koja je inicirala izuzetak, ne treba da se kompletiraju. Precizni izuzetak kod dinamičkog protočnog sistema nalaže da se očuva sekvencijalno ažuriranje arhitekturnog stanja i pored toga što se instrukcije izvršavaju *van-redosledno*.

Kod dinamičke protočne obrade instrukcije se pribavljaju i dekodiraju u programskom redosledu, ali se izvršavaju van programskog redosleda. U suštini instrukcije(a) ulaze(i) u rezervacione(u) stanice(u) u programskom redosledu, ali napuštaju(a) ih van redosleda. One takodje završavaju sa izvršenjem van redosleda. Da bi podržao rad preciznog izuzetka, kompletiranje instrukcije mora da bude u programskom redosledu tako da se ažuriranje arhitekturnog stanja mašine obavlja u programskom redosledu. Sa ciljem da se prilagodi *van-redosledni* završetak faze *execute* sa redoslednim završetkom faze *completion* instrukcije, neophodno je kod paralelnog protočnog sistema ugraditi bafer-preuredjenja (*reorder buffer*) u stepenu *completion*. Kako instrukcije završavaju sa izvršenjem one *van-redosledno* ulaze u bafer preuredjenja, ali izlaze iz njega u programskom redosledu. Nakon što izadju iz bafera preuredjenja, za njih smatramo da su arhitekturno kompletirane. Ovo je prikazano na slici 3.17, gde rezervacione stanice i bafer preuredjenja predstavljaju gornju i donju granicu *van-redoslednog* regiona protočnog sistema, tj. dela *Execute* protočnog sistema. Termini koji se koriste u ovom tekstu, a odnose se na različite faze procesiranja instrukcije prikazani su na slici 3.17.



Slika 3.17

Precizni izuzetak se obradjuje od strane stepena za kompletiranje instrukcija uz pomoć bafera za preuredjenje. Kada se javi izuzetak instrukcija koja je inicirala izuzetak se markira u beferu za preuredjenje. Stepen za kompletiranje instrukcija (*completion*) pre nego što kompletira svaku od instrukcija proverava. Kada se markirana (*targged*) instrukcija detektuje, njoj se ne dozvoljava kompletiranje. Svim sekvencijalnim instrukcijama, koje se u programskom redosledu nalaze pre markirane instrukcije, je dozvoljeno kompletiranje. Stanje mašine se zatim proverava ili pamti. Stanje mašine uključuje sve arhitekturne registre kao i programski brojač PC. Ostale instrukcije u protočnom sistemu, od kojih su neke već završile sa izvršenjem moraju biti anulirane. Nakon opsluživanja izuzetka markirano stanje mašine se obavlja, a izvršenje nastavlja sa pribavljanjem instrukcije koja je izazvala izuzetak.

4. SUPERSKALARNE TEHNIKE

Bolje performanse kod superskalarnih procesora se mogu postići efikasnijom propusnošću u toku procesiranja instrukcija. Procesiranje instrukcija treba sagledati kroz sledeća tri protoka:

- protok instrukcija (*instruction flow*): procesiranje *Branch* instrukcije
- protok registarskih podataka (*register data flow*): procesiranje *ALU* instrukcije
- protok memorijskih podataka (*memory data flow*): procesiranje *Load/Store* instrukcije

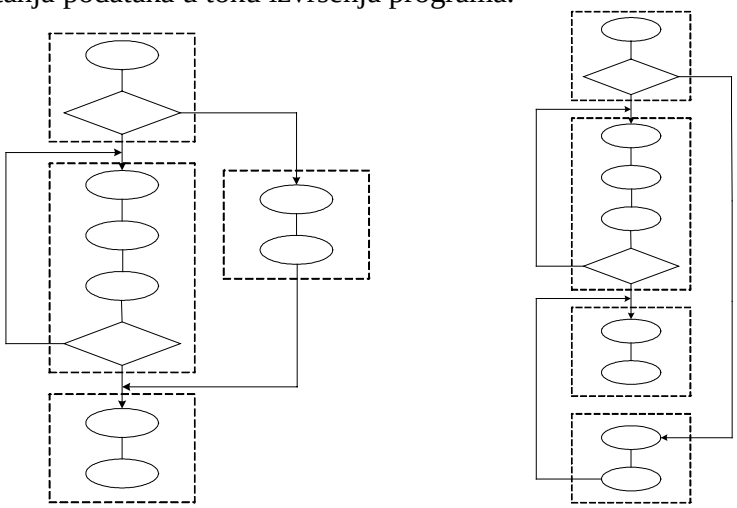
Sva tri protoka, u grubim crtama, odgovaraju procesiranju triju glavnih tipova instrukcija, *ALU*, *Branch* i *Load/Store*. Shodno prethodnom, maksimiziranje performansi sva tri protoka svodi se na minimiziranju cene koja se plaća usled procesiranja instrukcija tipa *Branch, ALU* i *Load/Store.*

4.1. Tehnike za povećanje protoka instrukcija

Efikasniji protok u procesiranju instrukcija, kod superskalarne protočne obrade, može se postići boljom organizacijom protočnih stepeni *Fetch* i *Decode*. Propusnost u radu ovih stepeni odredjuje gornju granicu propusnosti narednih stepeni u lancu protočne obrade. To znači da primarni cilj kod povećanja protoka instrukcija treba da je usmeren ka tome da se ostvari pravovremeno snabdevanje protočnog sistema instrukcijama.

4.1.1. Upravljanje tokom programa i kontrolne zavisnosti

Semantika upravljanja tokom programa specificira se u formi koja odgovara *grafu-toka-upravljanja* (*control flow graph*-CFG). Kod CFG-a čvorovi predstavljaju osnovne blokove, a potezi odgovaraju prenosu toka upravljanja izmedju osnovnih blokova. Na Slici 4.1 prikazan je CFG koga čine četiri osnovna bloka (pravougaonici uokvireni isprekidanim linijama), pri čemu se svaki blok sastoji od po nekoliko instrukcija (ovali). Usmereni potezi odgovaraju toku upravljanja izmedju osnovnih blokova, i ukazuju na egzistenciju uslovnih instrukcija grananja (rombovi). Izvršenje programa podrazumeva dinamički prolaz kroz čvorove i potege CFG-a. Startni prolazak duž nekog puta diktiran je instrukcijama tipa *Branch* kao i uslovima grananja (*branch condition*) koji su zavisni od trenutnog stanja podataka u toku izvršenja programa.



a) **b)**

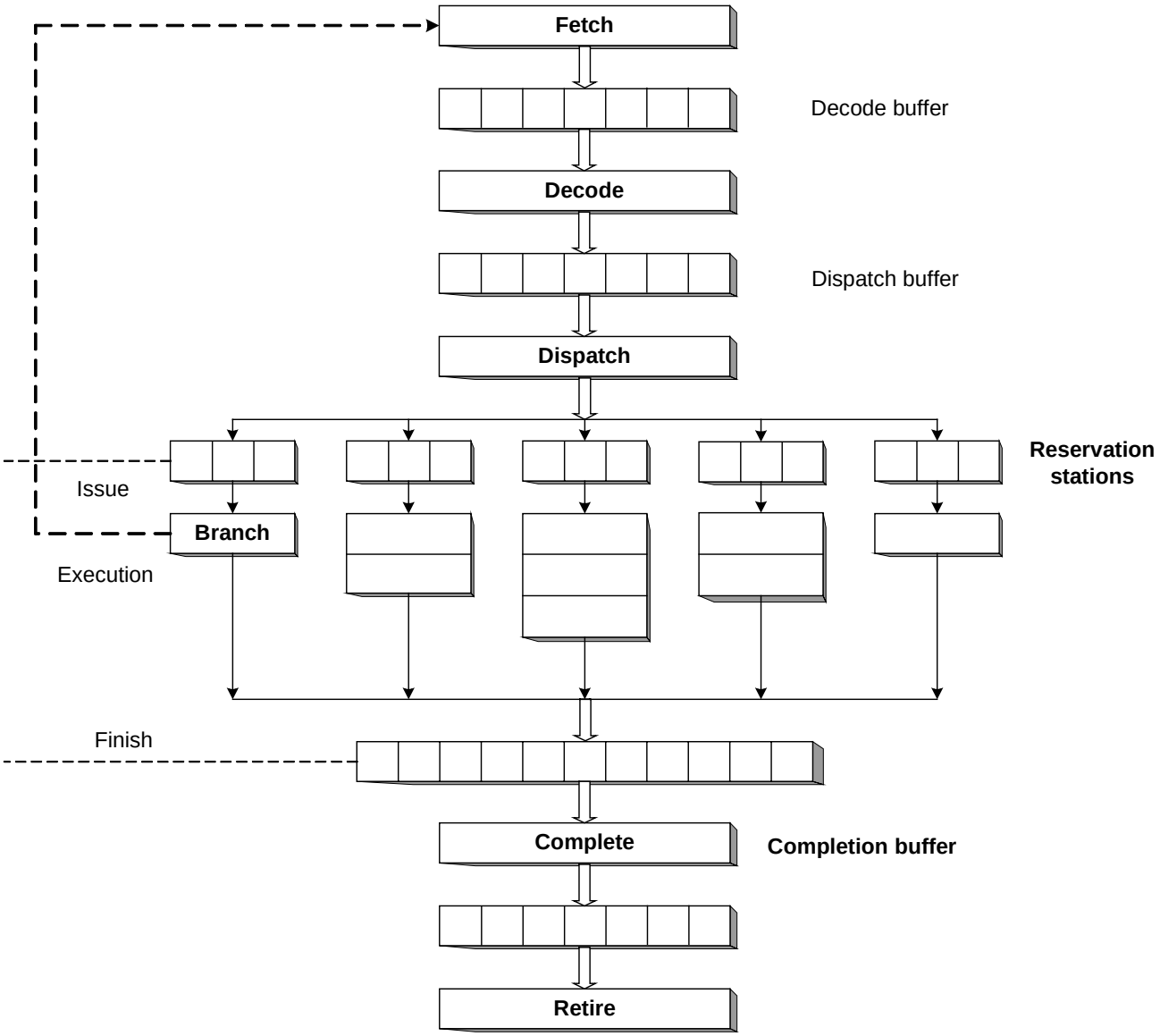
Slika 4.1 Upravljanje tokom programa: (a) graf toka upravljanja (CFG); (b) mapiranje (preslikavanje) CFG-a u sekvencijalne memorijske lokacije

Osnovni blokovi CFG-a (koji se sastoje od odgovarajućih pripadajućih instrukcija) moraju u programskoj memoriji biti smešteni u sekvencijalnim lokacijama. To znači da se parcijalni redosled osnovnih blokova CFG-a mora u programskoj memoriji urediti u formi koja je adekvantna potpunom (*total*) redosledu. U fazi preslikavanja CFG-a u linearne uzastopne memorijske lokacije, neophodno je, kako je to prikazano na slici 4.1, dodatno umetnuti bezuslovne instrukcije grananja (*unconditional Branch*). Preslikavanje (mapiranje) CFG-a u linearnu programsku memoriju u toku izvršenja programa odslikava posredno uredjen sekvencijalni tok upravljanja duž sekvencijalnih memorijskih lokacija. Ipak egzistencija uslovnih i bezuslovnih instrukcija grananja ukazuju na narušavanje sekvencijalnog toka izvršenja programa, a saglasno tome i odstupanje od sekvencijalnog načina pribavljanja instrukcija. Ova odstupanja uzrokuju zastoje u radu protočnog stepena za pribavljanje instrukcija i redukuju ukupnu propusnost. Instrukcije za poziv potprograma i povratak (*Callsubroutine i Return*) uzrokuju slične zastoje u sekvencijalnom pribavljanju instrukcija.

4.1.2. Degradacija performansi zbog grananja

Protočno organizovana mašina ostvaruje maksimalnu propusnost ako radi u *streaming* režimu (modu) rada. Za stepen *Fetch* ovakav režim rada se odnosi na pribavljanje instrukcija sa sekvencijalnih lokacija u programskoj memoriji. Kad god tok upravljanja programom odstupa od sekvencijalnog puta, dolazi do potencijalnog narušavanja *streaming* moda. Kod bezuslovnih instrukcija grananja, sve dok se ne odredi ciljna adresa grananja ne mogu se pribavljati naredne instrukcije. Kod uslovnih instrukcija grananja mora prvo da se sačeka trenutak dok se ne razreši dilema oko ispunjenja uslova grananja, a zatim ako do grananja dodje, mora da se čeka sve dok ciljna adresa grananja ne postane dostupna. Na slici 4.2 prikazano je kako se vrši narušavanje *streaming* moda od strane instrukcija tipa *Branch* i pokazano da se ovaj tip instrukcija izvršava od strane funkcionalne jedinice Branch. Kod uslovnih instrukcija grananja stepen *Fetch* može korektno da pribavlja nove instrukcije prvo kada se razreši dilema oko uslova grananja, a zatim kada se izračuna (odredi) ciljna adresa grananja.

Kašnjenje u procesiranju uslovnih instrukcija grananja uzrokuju povećanje cene od tri ciklusa kod pribavljanja naredne instrukcije, a to odgovara prolasku instrukcije uslovno grananje kroz stepene *Decode, Dispatch* i *Execute*. Stvarnu cenu koju treba platiti, a ogleda se u gubitku od tri ciklusa, u suštini ne iznosi tri prazna slota instrukcija, kakav je to slučaj kod skalarnih procesora, nego se taj broj mora pomnožiti sa obimom mašine, tj. stepenom superskalarnosti. Tako na primer ako je stepen superskalarnosti $n = 4$ (u jednom ciklusu se pribavljaju po četiri instrukcije) tada ukupna cena koja se plaća iznosi 12 mehurova. Ovakvi zastoji u radu u značajnoj meri smanjuju stvarne performanse koje se mogu ostvariti.



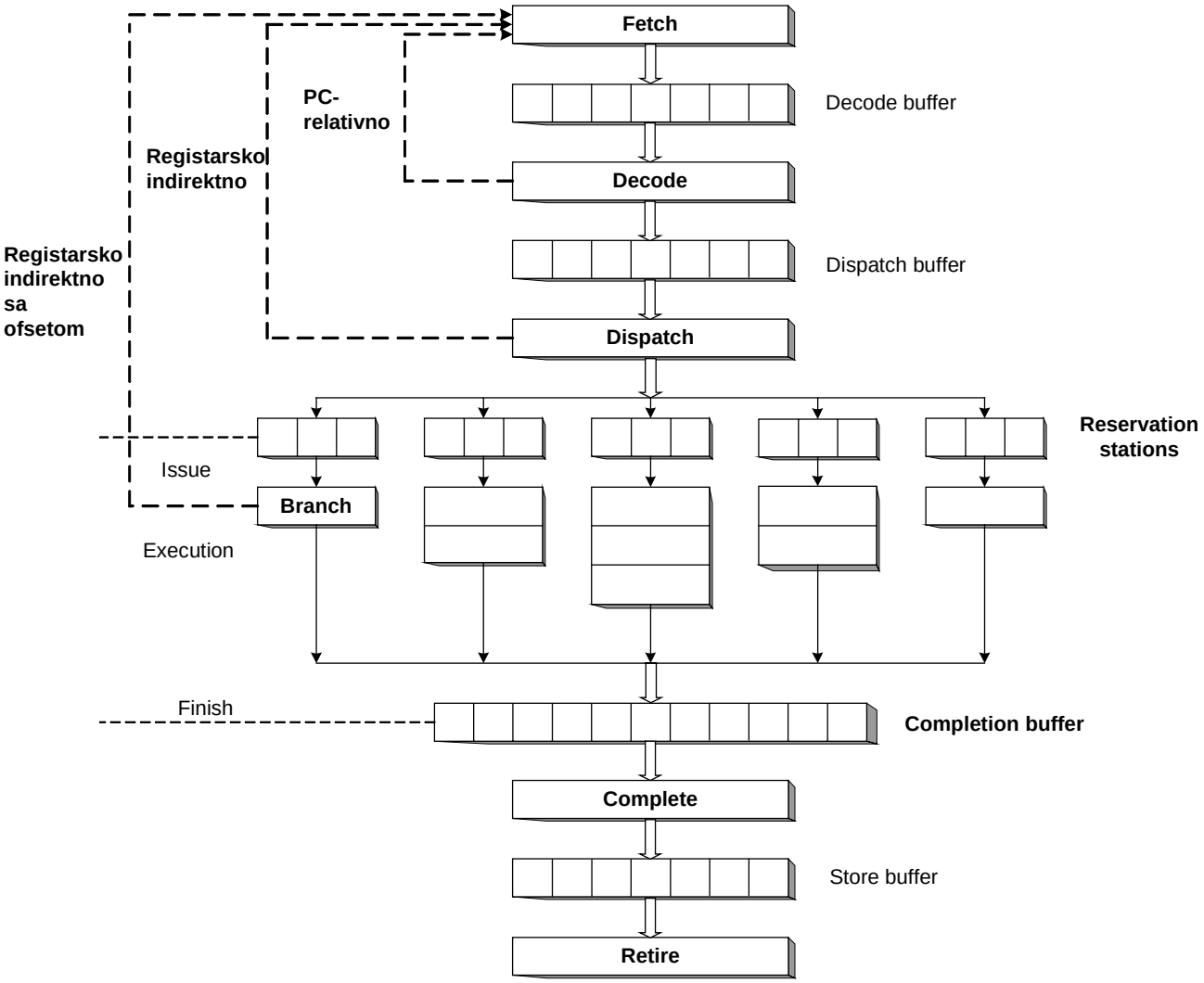
Slika 4.2 Narušavanje sekvencijalnog toka upravljanja od strane instrukcije tipa *Branch*

Kod uslovnih instrukcija grananja, stvarni broj zastoja, ili cena koja se plaća u zastojima, diktirana je od strane generisanja ciljne adrese grananja ili razrešavanja dileme u vezi grananja. Stvarni broj ciklusa zastoja odredjen je od strane adresnih načina rada koje koriste instrukcije tipa *Branch*.

Kod PC-relativnog adresnog načina rada, ciljna adresa grananja se može generisati u stepenu *Fetch*, što rezultira ceni zastoja od jednog ciklusa.

Ako se koristi registarsko indirektni adresni način rada, tada instrukcija *Branch* mora da prodje stepen *Decode* kako bi se pristupilo registru. U ovom slučaju cena koja se plaća zbog zastoja iznosi dva ciklusa.

Kada se koristi registarsko indirektno adresiranje sa ofsetom, ofset se mora dodati nakon pristupa registru tako da ukupni broj zastoja iznosi tri ciklusa. Na slici 4.3 prikazan je potencijalni broj ciklusa zastoja koji se javljaju zbog generisanja ciljne adrese grananja.



Slika 4.3 Cena koja se plaća zbog generisanja ciljne adrese instrukcije tipa *Branch*

Različite metode kojima se razrešavaju dileme vezane za uslov grananja rezultiraju različitim cenama, izražene u broju zastoja, koje se moraju platiti. Za slučaj da se koriste CCR-ovi (*Condition Code Registers*) i ako usvojimo da se relevantnom CCR-u pristupa u stepenu *Dispatch*, tada cena zastoja iznosi dva ciklusa. Ako ISA, da bi generisala uslov grananja, generisanje uslova ostvaruje uporedjivanjem sadržaja dva registra opšte namene, tada je neophodan još jedan ciklus kako bi se obavila ALU operacija nad sadržajem oba registra. To će rezultirati ceni zastoja od tri ciklusa. Kod uslovne instrukcije grananja, u zavisnosti od adresnog načina rada i pristupa u razrešavanju dileme grananja, jedan od oba pomenuta uzročnika je kritičan. Tako na primer, čak i da se koristi PC-relativni adresni način rada, uslovna instrukcija grananja mora da pristupi CCR-u, a to kod izračunavanja ciljne adrese grananja rezultuje ceni zastoja od dva, a ne jednog ciklusa.

Maksimiziranje iznosa dužine puta koji se odnosi na *instrukcioni-tok* direktno odgovara maksimiziranoj propusnosti kod pribavljanja instrukcija. Naglasimo da je :

ukupna cena gubitaka srazmerna proizvodu broja ciklusa – zastoja

i širine $\left(\textit{stepena sup erskalarnosti}\right)$ *mašine*

Kod mašine čiji je stepen superskalarnosti n svaki ciklus zastoja ekvivalentan je pribavljanju (ubacivanju) n operacije tipa *Nop*. Glavni cilj tehnike za povećanje protoka instrukcija je da minimizira

broj ciklusa zastoja, i/ili da u toku tih ciklusa obezbedi uslove da se potencijalno obavi neki koristan posao. Tekuće dominantni pristup da se ostvari ovaj cilj predstavlja pristup koji se bazira na tehnici predikcije (predvidjanja) grananja (*branch prediction*).

4.1.3. Tehnike za predikciju grananja

Do sada sprovedena ispitivanja vezana za osobine kod izvršavanja programa pokazuju da se ponašanje instrukcija tipa *Branch* može dosta dobro i pouzdano predvideti. Ključni pristup u minimiziranju cene koja se mora platiti zbog grananja, a shodno tome maksimiziranju protoka instrukcija, bazira se na spekulativnom manipulisanju, prvo one koja se odnosi na izračunavanja vrednosti ciljne adrese grananja, a drugo one koja se tiče procene uslova grananja. Kada se jedna statička instrukcija repetitivno izvršava u vremenu, tada se njeno dinamičko ponašanje može pratiti. Na osnovu njenog ponašanja u prošlosti, moguće je efikasno predvideti njeno ponašanje u budućnosti. Dve osnovne komponente predikcije grananja su:

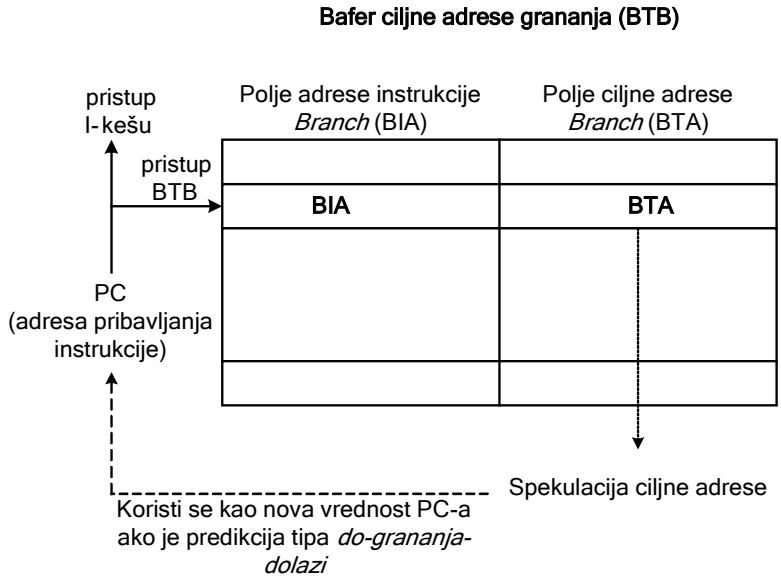
1. spekulacija sa ciljnom adresom grananja (*branch target speculation*)
2. spekulacija u vezi uslova grananja (*branch condition speculation*)

Svaku spekulativnu tehniku prati odgovarajući mehanizam koji se odnosi na validaciju predikcije, ali i mehanizam za bezbedno oporavljanje u slučaju kada je predikcija pogrešna.

Spekulacija sa ciljnom adresom grananja podrazumeva korišćenje bafera ciljne adrese grananja (*Branch Target Buffer-BTB*) u kome se čuvaju prethodne ciljne adrese grananja. *BTB* predstavlja malu keš memoriju kojoj se pristupa u toku stepena za pribavljanje instrukcija koristeći tekuću vrednost PC-a. Svaki ulaz (vrsta) *BTB*-a sadrži dva polja:

1. adresa instrukcije grananja (*branch instruction address-BIA*) i
2. ciljna adresa grananja (*branch target address-BTA*).

Kad se statička *Branch* instrukcija izvršava po prvi put, dodeljuje joj se ulaz i *BTB.* Adresa *Branch* instrukcije se smešta u *BIA* polje, a ciljna adresa grananja u *BTA* polje. Ako usvojimo da je *BTB* potpuno asocijativni keš, tada se *BIA* polje koristi za asocijativni pristup *BTB*-u. U suštini *BTB*-u se pristupa konkurentno sa pristupom I-kešu. Kada se tekuća vrednost PC-a upari sa *BIA* kao ulaz u *BTB*, kažemo da dolazi do pogotka kod *BTB*-a. To znači da je tekuća instrukcija koja se pribavlja iz I-keša prethodno već bila izvršena i da je to instrukcija tipa *Branch*. Kada dodje do pogodtka, pristupa se BTA polju koje je pridruženo ulazu za koji postoji pogodak, pa se adresa koja se pročita sa tog polja koristi kao adresa za pribavljanje naredne instrukcije, naravno pod uslovom da je predikcija bila tipa *do-grananja-dolazi* (vidi sliku 4.5).



Slika 4.5 Spekulacija kod ciljne adrese grananja koristeći *BTB*

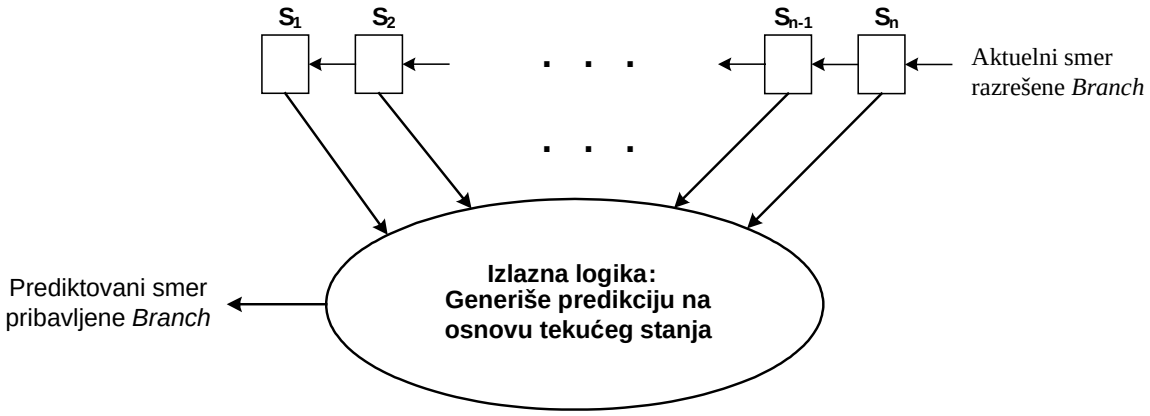
Pristupanjem tabeli *BTB* koristeći adresu instrukcije grananja i izbavljanjem ciljne adrese grananja iz *BTB*-a, u toku stepena *Fetch*, spekulativna ciljna adresa grananja postaće spremna za korišćenje u narednom mašinskom ciklusu, i to kao adresa sa koje se pribavlja nova instrukcija pod uslovom da *do-grananja-dolazi*. Ako je predikcija bila tipa da *do-grananja-dolazi* i ustanovi se kasnije da je ona bila korektna, tada se instrukcija *Branch* efektivno izvršava u stepenu *Fetch*, što rezultira da ne postoje zastoji, tj. ne treba platiti cenu zbog zastoja. No nespekulativno izvršenje instrukcije *Branch* se i dalje obavlja sa ciljem da se validira (potvrdi) spekulativno izvršenje. Instrukcija *Branch* se pribavlja iz I-keša i izvršava. Ciljna adresa grananja kao i uslov grananja se proveravaju u odnosu na spekulativnu verziju. Ako postoji slaganje, tada za predikciju kažemo da je bila korektna, u ostalom slučaju, predikcija je bila pogrešna tako da je neophodno inicirati akciju oporavljanja. Rezultat nespekulativnog izvršenja se takodje koristi za ažuriranje sadržaja, tj. *BTA* polja u okviru *BTB*-a.

Postoje brojni načini koji se koriste za spekulativnu predikciju grananja. Kod najjednostavnijeg rešenja *Fetch* hardver se projektuje tako da *do-grananja-ne-dolazi*, tj. predikcija je *grananje-se-ne-preduzima*. Kada se u toku izvršenja naidje na instrukciju *Branch*, pre nego što se dilema oko grananja razreše, stepen *Fetch* produžava bez zastoja da pribavlja one instrukcije koje u sekvencijalnom programskom redosledu neposredno slede nakon instrukcije *Branch*. Ovaj oblik minimalne predikcije grananja je lak za implementaciju, ali nije efikasan. Tako na primer, veliki broj instrukcija grananja se koristi kao zadnja instrukcija u petlji, a kod ovakvih situacija u najvećem broju slučajeva do grananja dolazi, osim u zadnjoj iteraciji kada se izlazi iz petlje.

Jedno drugo rešenje predikcije se bazira na korišćenju softverske podrške, ali zahteva promene u *ISA*. Na primer, ekstra bit koji se postavlja od strane kompilatora se može pridružiti formatu instrukcija *Branch*. Na osnovu vrednosti na koju je postavljen ovaj bit hardver koristi predikciju do *grananja-ne-dolazi* ili *do-grananja-dolazi*. Kompajler (može da) koristi instrukciju tipa *Branch* i profiliše informaciju sa ciljem da odredi vrednost koja najviše odgovara ovom bitu. Ovakvim pristupom se obezbedjuje da svaka instrukcija *Branch* ima svoju specificiranu predikciju. Ipak, ova predikcija je statička, u smislu što se za sva dinamička izvršenja instrukcije tipa *Branch*, koristi isto predvidjanje. Ovakav tip statičke softverske predikcije se koristi kod procesora *Motorola M 88 110*. Kod trećeg rešenja, koristi se agresivna i dinamička forma predikcije koja sve bazira na korišćenju ofseta ciljne adrese grananja. Kod ove forme predikcije prvo se odredjuje relativni ofset izmedju adrese instrukcije *Branch* i adrese ciljne instrukcije. Pozitivni ofset trigeruje hardver da primeni predikciju *do-*

grananja-ne-dolazi, dok negativni ofset, u najvećem broju slučajeva odgovara zadnjoj instrukciji tipa *Branch* u petlji, trigerovaće hardver da obavi predikciju *do-grananja-dolazi*. Ova tehnika, bazirana na *Branch* ofsetu koristi se kod procesora *IBM RS/6000*, a prihvaćena je kao dobro rešenje i kod drugih mašina. Kod savremenih superskalarnih mašina najčešće korišćena tehnika koja se odnosi na spekulaciju uslova grananja je ona koja se bazira na istoriji prethodnih izvršenja grananja.

Na osnovu prethodnih opservacija o smeru grananja, tehnika za *predikciju-grananja-bazirana-na-istoriji* (*history based branch prediction*) donosi odluke o smeru grananja u formi grananje *postoji* (*taken -T*) ili *ne-postoji* (*not taken-N*). Pretpostavka se zasniva na tome da informaciji o istoriji smera grananja u toku prethodne iteracije može biti od velike koristi kod odredjivanja uslova grananja u narednoj iteraciji. Projektantske odluke kod ovakvog tipa predikcije grananja uzimaju u obzir koliko dugo (misli se na broj iteracija) se vodi evidencija o istoriji, i u toku observiranog istorijskog perioda kakav je bio ishod predikcije (koliko puta je bio pogodak, a koliko promašaj). Specifični algoritam koji se odnosi na predikciji smera grananja zasnovan (baziran**)** na istoriji se može opisati (vidi sliku 4.6) koristeći princip rada konačnog automata (*Finite State Machine-FSM*). Na osnovu *n* promenljivih stanja kodiraju se smerovi grananja uzimajući u obzir zadnjih *n* izvršenja tog grananja. To znači da se svako stanje odnosi na pojedinu istorijsku formu koja zavisi od toga da li je do grananja došlo ili nije došlo. Izlazna logika generiše predikciju na osnovu tekućeg stanja FSM-a. U suštini, predikcija se formira na osnovu ishoda prethodnih *n*-izvršenja tog grananja. Kada se prediktovano grananje konačno obavi, stvarni ishod se koristi kao ulaz u FSM radi trigerovanja prelaska iz jednog stanja u drugo. Logika *naredno-stanje* je trivijalna, ona jednostavno ulečava (smešta) u pomerački registar promenljive stanja, čime zapisuje (vodi evidenciju) o smeru grananja instrukcije *Branch* za njena prethodna *n* izvršenja.

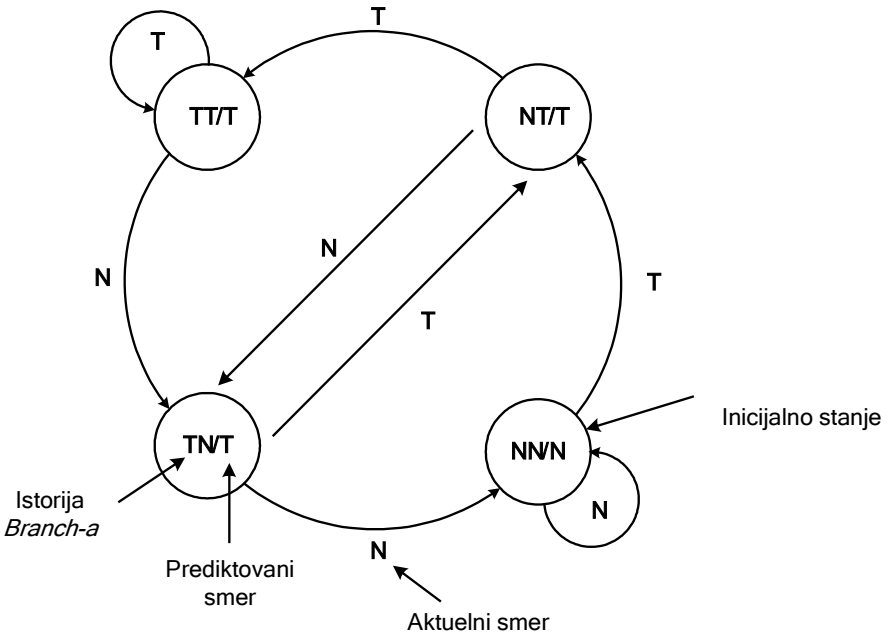


Slika 4.6 *FSM* model prediktora smera grananja *zasnovan-na-istoriji*

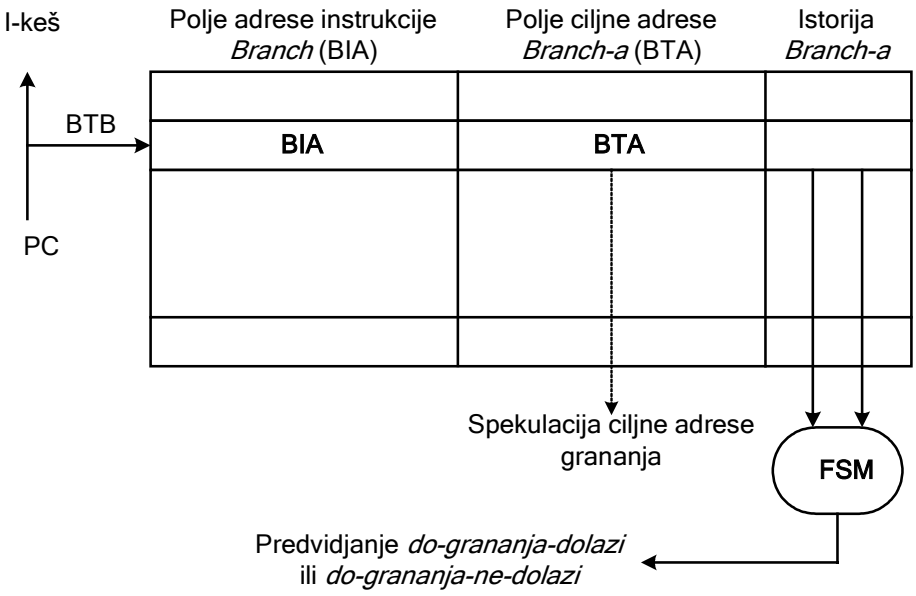
Na Slici 4.7 prikazan je *FSM* dijagram jednog tipičnog 2-bitnog *Branch* prediktora koji koristi dva-bita istorije kako bi pratio ishod prethodna dva izvršenja instrukcije *Branch*. Dva bita istorije predstavljaju promenljive stanja *FSM*-a. Prediktor se može naći u jedno od četiri stanja: *NN, NT, TT* i *TN*, što odgovara preduzetim smerovima grananja u toku prethodna dva izvršenja instrukcije *Branch*. Inicijalno se dodeljuje stanje *NN*. Izlazna vrednost *T* ili *N* se dodeljuje svakom od četiri stanja, a to odgovara preduzetom predvidjaju kada se prediktor nadje u to stanje. Kada se obavlja grananje, aktuelno preduzeti smer grananja se koristi kao ulaz u *FSM*, a samim prelaskom iz jednog stanja u drugo se ažurira vrednost istorije grananja koja se koristi u toku donošenja odluke kod narednog predvidjanja.

Implementirani algoritam prediktora sa slike 4.7 orijentisan je ka predikciji grananja kojom se uvek predvidja da *do-grananja-dolazi*; uočimo da se kod tri od četiri stanja predvidja da će *do-grananja-doći*. On polazi od toga da će se dugo ponavljati smer grananja *N*(*T*) ako je *FSM* u stanju *NN*

(*TT*). Za slučaj da je u toku jednog od prethodna dva izvršenja došlo do grananja prediktovani smer je da do grananja dolazi. Predikcija komutiraće kada do grananja ne dolazi ako se zaredom jave dva *N*-a. Algoritam sa slike 4.7 a) predstavlja samo jedan od brojnih algoritama.



a)



b)

Slika 4.7 Predikcija grananja zasnovana na istoriji; a) Algoritam 2-bitnog *Branch* prediktora; b) *BTB* sa dodatnim poljem za memorisanje bitova o istoriji *Branch*-a

Da bi se podržao rad prediktora smera grananja zasnovan na istoriji, svaki ulaz *BTB*-a treba proširiti poljem istorije. Obim, broj bitova, ovog polja odredjen je brojem bitova istorije koje treba

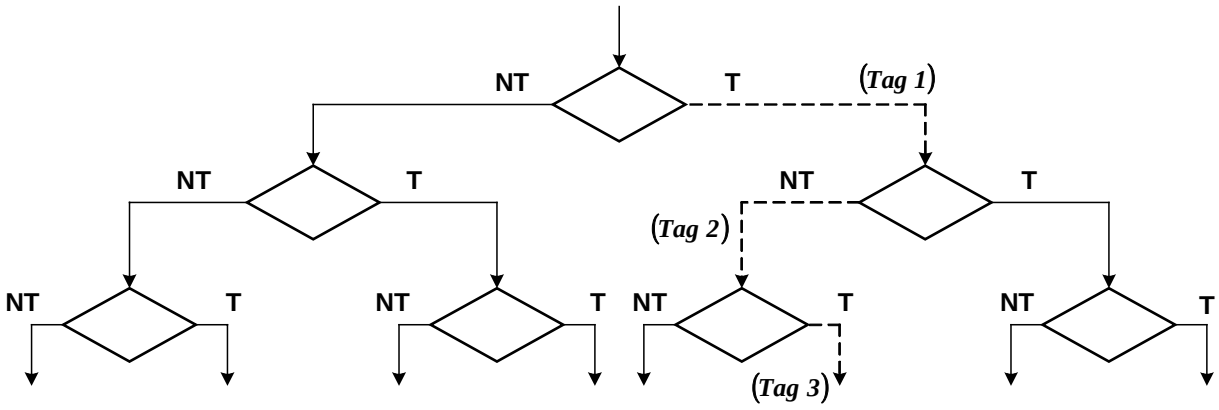
pratiti. Kada se PC adresa dovede na ulaz *BTB*-a, pored spekulativne ciljne adrese grananja, iz tabele se čitaju i bitovi istorije. Bitovi istorije se dovode na ulaz logike koja kod *FSM Branch* prediktora implementira funkcije *naredno-stanje* i izlaz. Bitovi istorije koje smo dobavili iz tabele koriste se kao promenljive stanja (ulazne promenljive) *FSM*-a. Na osnovu bitova istorije izlazna logika generiše 1-bitni izlaz koji ukazuje na predvidjeni smer grananja. Ako se predikcija odnosi na slučaj kada *do-grananja-dolazi*, tada se izlaz koristi da usmeri spekulativnu ciljnu adresu grananja ka PC-u kako bi se u narednom ciklusu pribavila nova instrukcija. Za slučaj da je predikcija grananja bila korektna, tada se instrukcija *Branch* efektivno izvršava u stepenu *Fetch*, a cena koja se plaća zbog zastoja je nula mašinskih ciklusa.

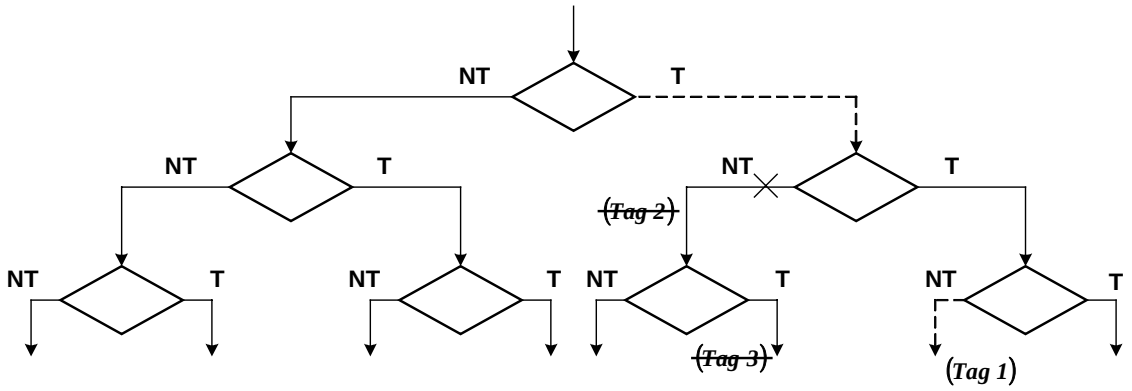
Godine 1984. *Lee* i *Smith* napravili su eksperimente koji se odnose na predikciju grananja. Na tri različite mašine, *IBM 370, DEC PDP-11* i *CDC 6400*, izvršavali su 26 programa koji su pripadali 6 različitim radnim opterećenjima. U proseku kod sva šest benčmarka (*benchmark*) u ***67,6 %*** slučajeva do grananja je dolazilo, a u ***32,4 %*** nije dolazilo, tj. odnos smerova grananja je bio dva na prema jedan. Sa statičkom predikcijom grananja zasnovano na *op-kôd-tipu*, za sva šest radna opterećenja tačnost predvidjanja je bila u granicama od ***55 %*** do ***85 %***.

Korišćenjem samo jednog bita istorije, kod dinamičke predikcije grananja postignuta je tačnost predvidjanja u opsegu od ***79,7 %*** do ***96,5 %***. Kod prediktora za dva bita istorije tačnost predvidjanja je bila u granicama od ***83,4 %*** do ***97,5 %***. Povećanjem broja bitova istorije raste i stopa pogodaka, ali treba naglasiti da su za broj bitova istorije veći od 4 dodatni inkrementi u tačnosti predvidjanja manja.

4.1.4. Oporavljanje nakon pogrešne predikcije kod grananja

Predikcija grananja predstavlja spekulativnu tehniku. To drugim rečima znači da su joj, kao bilo kojoj spekulativnoj tehnici, potrebni mehanizmi za validaciju spekulacije. Shodno ovoj konstataciji dinamičku predikciju grananja možemo posmatrati kao proces koji se izvršava od strane dve interaktivne mašine (vodeća i prateća) . Vodeća mašina obavlja spekulaciju na *front-end* stepenima kod protočne obrade, dok prateća mašina obavlja validaciju kod kasnijih stepena u protočnoj obradi. U slučaju pogrešne predikcije kod grananja kasniji stepeni su ti koji su zaduženi da ostvare oporavljanje. Ova dva aspekta predikcije grananja prikazana su na slici 4.9.





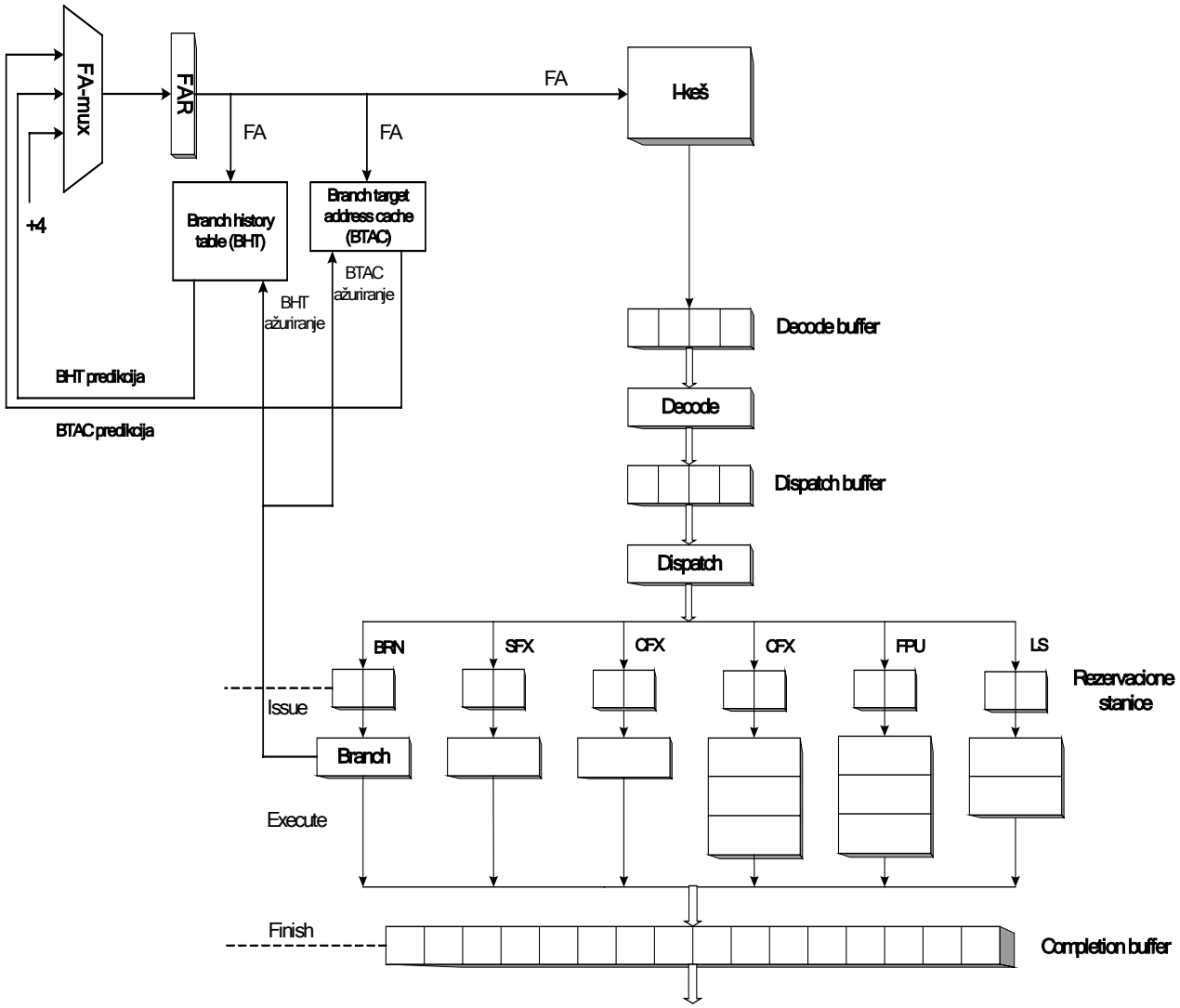
Slika 4.9 Dva aspekta predikcije *Branch*-a: a) spekulacija kod *Branch*-a; b) validacija/oporavljanje kod *Branch*-a

Spekulacija grananja (*branch speculation*) se sastoji od predikcije smera grananja nakon čega sledi pribavljanje instrukcija sa prediktovanog puta toka upravljanja. U toku pribavljanja instrukcija sa prediktovanog puta, moguće je da se ponovo naidje na instrukcije grananja. Predikcija (ovih) dodatnih grananja može se na sličan način obaviti kao i predikcija prve, a to rezultira spekulativnom prolazu većeg broja uslovnih instrukcija grananja, pri čemu dileme oko prve spekulacije još nisu razrešene. Na slici 4.9 a) prikazana su tri spekulativna prolaza instrukcija grananja pri čemu je prvo i treće grananje prediktovano kao *do-grananja-dolazi* (*branch taken –T*), a drugo kao *do-grananja-ne-dolazi* (*branch not taken –NT*). Kada dodje do ovakve situacije, instrukcije iz sva tri spekulativna bloka su rezidentne (nalaze se) u mašini i moraju biti identifikovane (označene) na odgovarajući način. Pri ovome, instrukcijama koje pripadaju jednom spekulativnom bloku se dodeljuje isti (jedinstveni) identifikujući marker (*tag*). Tako na primer, kako je to prikazano na slici 4.9 a), za identifikaciju instrukcija koje pripadaju trima različitim spekulativnim osnovnim blokovima dodeljuju se tri različita markera (*tag*-ova). Pri ovome, označena (*tagged*) instrukcija ukazuje da je to spekulativna instrukcija, dok vrednost markera identifikuje kom osnovnom bloku ta instrukcija pripada. Kako spekulativna instrukcija avansuje duž protočnih stepeni, sa njom se prenosi i marker. Kod spekulacije, adrese instrukcija svih spekulativnih instrukcija grananja (ili narednih sekvencijalnih instrukcija) moramo baferovati (zapamtiti), za slučaj da je potrebno sprovesti postupak oporavljanja.

Validacija grananja (*branch validation*) se javlja u toku izvršenja grananja kao i razrešavanja dileme oko aktuelnog smera grananja. To znači da je moguće odrediti korektnost ranije predikcije. Ako se za predikciju pokaže da je bila korektna, marker spekulacije se briše (dealocira) i svim instrukcijama kojima je taj marker pridružen dozvoljeno je kompletiranje. U slučaju kada se detektuje pogrešna predikcija preduzimaju se dve akcije. Prvo, zatvara se (zaustavlja se) nekorektni put, a drugo, inicira se pribavljanje instrukcija sa novog korektnog puta. Da bi se inicirao novi put, neophodno je ažurirati vrednost PC-a na adresu nove instrukcije. Za slučaj da je nekorektna predikcija bila *T* (*branch taken*) tada se PC ažurira na adresu sekvencijalne instrukcije, koja se odredjuje na osnovu adrese prethodno baferovane instrukcije kada je grananje bilo prediktovano kao *T*. Nakon ažuriranja PC-a, nastavlja se pribavljanje instrukcija sa novog puta, a predikcija grananja počinje iznova. Za zatvaranje nekorektnog puta, koriste se markeri spekulacije. Svi markeri koji prate (pridruženi su) pogrešno prediktovano grananje koriste se za identifikaciju instrukcija koje moraju biti eliminisane. Sve te instrukcije koje se nalaze u baferima za dekodiranje i rasporedjivanje (*dispatch*) kao i one koje se nalaze u ulazima rezervacionih stanica neophodno je invalidirati. Ulazi bafera preuredjenja (*reorder buffer*) koji su zauzeti (koji se pamte) od strane ovih instrukcija moraju se dealocirati (osloboditi). Na slici 4.9 b) prikazan je proces *validacija/oporavljanje* za slučaj kada je druga od tri predikcije bila pogrešna. Prvo grananje je bilo korektno prediktovano, pa zbog toga, instrukcije kojima je pridružen *Tag1* postaju nespekulativne i dozvoljava im se kompletiranje. Druga predikcija je bila nekorektna, tako da sve

instrukcije kojima su pridruženi markeri *Tag2* i *Tag3* moraju biti invalidirane, a njihovi ulazi u bafer preuredjenja moraju se dealocirati (anulirati). Nakon pribavljanja informacije o novom putu, predikcija grananja može da počne iznova, pri čemu se *Tag1* može ponovo koristiti da ukaže na instrukcije koje pripadaju prvom spekulativnom osnovnom bloku. U toku validacije grananja, odgovarajući (pridruženi) *BTB* ulaz se takodje ažurira.

Sada ćemo koristiti Power PC 604 superskalarni mikroprocesor da bi ilustrovali implementaciju dinamičke predikcije grananja kod jednog realnog superskalarnog procesora. Power PC je superskalarni procesor obima 4 koji je u stanju da pribavlja, dekodira i dispečuje do četiri instrukcije svakog mašinskog ciklusa. Umesto jedinstvenog *BTB*-a, Power PC 604 koristi dva izdvojena bafera koji podržavaju predikciju grananja, naime adresni keš ciljnog grananja (*branch targget address cashe-BTAC*) i tabela istorije grananja (*branch history table-BHT*) vidi sliku 4.10. *BTAC* je potpuno asocijativni keš sa 64 ulaza u kome se pamte ciljne adrese grananja, dok *BHT* je direktno-preslikana tabela sa 512 ulaza u kojoj se čuvaju bitovi o istoriji grananja. Razlog ovom razdvajanju biće kratko objašnjen.



Slika 4.10 Predikcija grananja kod Power PC 604 superskalarnog mikroprocesora

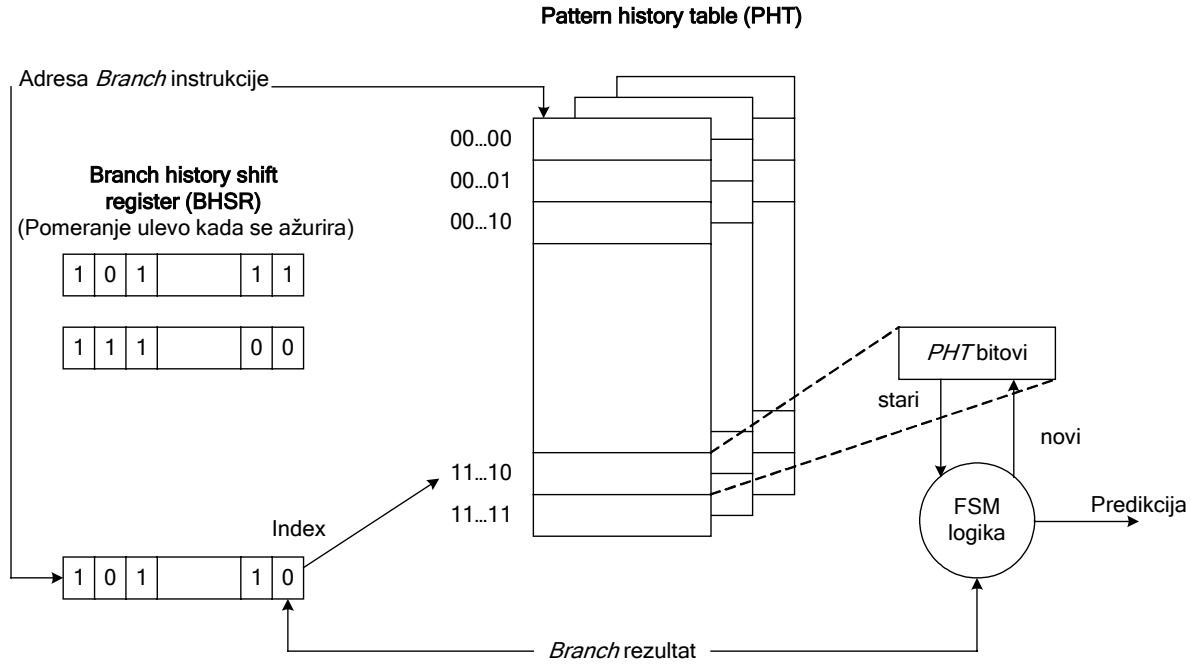
Napomena:: *Branch history table* (*BHT*)- Tabela istorije instrukcije *Branch*; *Branch target adrress cache*(*BTAC*)- Keš ciljne adrese grananja instrukcije *Branch*.

Obema tabelama, *BTAC* i *BHT*, se pristupa u toku stepena *Fetch* koristeći tekuću adresu za pribavljanje instrukcija koja se čuva u PC. *BTAC* se odaziva nakon jednog ciklusa, dok *BHT*-u su potrebna dva ciklusa za kompletiranje pristupa. Ako se javi pogodak u *BTAC*-u, koji ukazuje na prisustvo instrukcije grananja u tekućoj grupi za pribavljanje predikcija koja se odnosi na *do-grananja-dolazi* se javlja, a ciljna adresa grananja koja se izbavlja iz *BTAC*-a se koristi u narednom ciklusu pribavljanja. Pošto Power PC 604 pribavlja 4 instrukcije u ciklusu *Fetch*, moguće je da postoji veći broj grananja u grupi koja se pribavi. Zbog toga *BTAC* ulaz koji se indeksira od strane adrese za pribavljanje sadrži ciljnu adresu grananja prve *Branch* instrukcije u grupi koja se pribavlja, a koja je prediktovana kao *do-grananja-dolazi*. U drugom ciklusu, ili u toku stepena *Decode*, bitovi istorije koji se izbavljaju iz *BHT*-a koriste se za generisanje predikcije bazirane na istoriji, a odnose se na istu instrukciju *Branch*. Ako dodje do slaganja obeju predikcija za prvu predikciju kažemo da je važeća (*BTAC*). Sa druge strane, ako se *BHT* predikcija ne slaže sa *BATC* predikcijom, *BTAC* predikcija se anulira (poništava), a pribavljanje se ostvaruje sa drugog puta, što odgovara predikciji *do-grananja-ne-dolazi*. U suštini *BHT* predikcija je većeg prioriteta od *BTAC* predikcije. Kao što je zaočekivati, kod najvećeg broja slučajeva dolazi do slaganja obeju predikcija. U nekim slučajevima, *BHT* koriguje pogrešnu predikciju koju je učinio *BTAC*. Ipak je moguće da *BHT* pogrešno promeni korektnu predikciju *BTAC*-a, ali do ovakve situacije dolazi veoma retko. Kada se problem grananja razreši, *BHT* se ažurira, a na osnovu ažuriranog sadržaja *BHT*-a ažurira se i *BTAC*. Power PC 604 ima 4 ulaza u rezervacionu stanicu koji se dovode na ulaz *Branch Execution Unit* (*BEU*) zbog toga ovaj mikroprocesor može da spekuliše do 4 *Branch* instrukcija tj. istovremeno maksimalno postoje do 4 spekulativne *Branch* instrukcije u mašini. Da bi označili 4 spekulativna osnovna bloka, koristi se dvobitni marker radi identifikacije svih spekulativnuh instrukcija. Nakon što se grananje reši, vrši se validacija grananja pri čemu se sve spekulativne instrukcije mogu učiniti nespekulativnim ili se invalidiraju pomoću dvobitnog markera. Ulazi bafera preuredjenja koji se zauzimaju od strane pogrešno spekulisane instrukcije se dealociraju. Ponovo, ovo se obavlja pomoću dvobitnog markera.

4.1.5. Napredne tehnike za predikciju grananja

Šeme za dinamičku predikciju grananja o kojima smo prethodno diskutovali imaju veći broj ograničenja. Predikcija o grananju se donosi na osnovu ograničene istorije samo o toj statičkoj *Branch* instrukciji. Algoritam o aktuelnoj predikciji ne vodi računa o dinamičkom kontekstu u okviru koga se inatrukcija *Branch* izvršava. Tako na primer, algoritam ne koristi informaciju o pojedinom toku upravljanja kada naidje na tu instrukciju. Šta više isti fiksni algoritam se koristi za predikciju grananja nezavisno od dinamičkih uslova. Eksperimentalnom analizom je ustanovljeno da ponašanje odredjenih grananja strogo zavise od ponašanja drugih grananja koje mu u toku izvršenja prethode. Saglasno tome, može se ostvariti tačnije predvidjanje pomoću algoritama koje vode računa o istoriji grananja drugih korelisanih (srodnih) grananja, a da pri tome prilagode algoritam predikcije dinamičkom kontekstu grananja.

Yeh i *Patt* 1982. godine predložili su dvo-nivovsku adaptivnu tehniku za predikciju grananja koja može potencijalno da postigne tačniju predikciju grananja od 95% na osnovu veoma fleksibilnog algoritma za predikciju koji se prilagodjava promenama konteksta. Kod prethodnih šema, koristi se jedinstvena tabela o istoriji grananja koja se indeksira od strane *Branch* adrese. Za svaku *Branch* adresu postoji samo jedan relevantni ulaz u *BHT*-u. Kod dvo-nivovske adaptivne šeme koristi se skup tabela istorije. One se identifikuju kao *PHT*-ovi (*Pattern History Table*) vidi sliku 4.11. Svaka *Branch* adresa indeksira skup relevantnih ulaza; jedan od ovih ulaza se zatim selektuje na osnovu konteksta dinamičkog grananja. Kontekst se odredjuje na osnovu specifičnog oblika najskorije izvršenih grananja koji se čuva u *BHSR*-u (*Branch History Shift Register*) vidi sliku 4.11. Sadržaj *BHSR*-a se koristi za indeksiranje *PHT*-a radi selekcije jednog od relevantnih ulaza. Sadržaj ovog ulaza se nakon toga koristi kao stanje *FSM*-ovog algoritma za predikciju radi generisanja predikcije. Kada se grananje razreši, rezultat grananja se koristi za ažuriranje kako *BHSR*-a tako i selektovanog ulaza *u PHT*-u.



Slika 4.11 Dvo-nivovska adaptivna *Branch* predikcija po *Yeh* i *Patt*-u

Napomena:: *Pattern history table* (*PHT*)- Oblik tabele istorije; *Branch history shift register*(*BHSR*)- Pomerački registar o istoriji instrukcije *Branch*

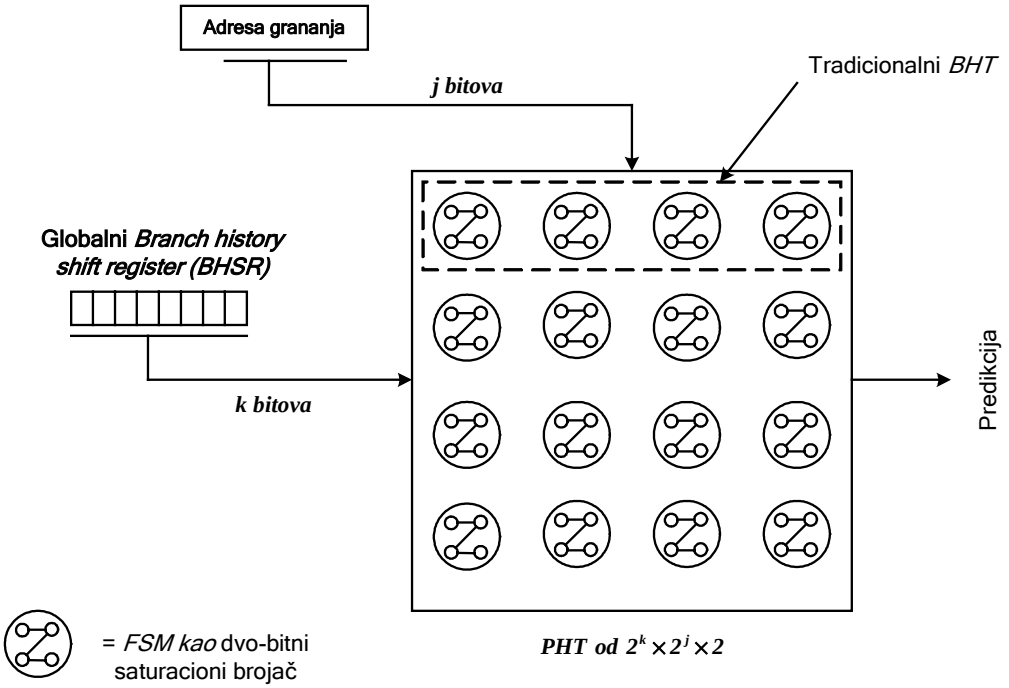
Dvo-nivovska adaptivna tehnika za predikciju grananja predstavlja okvir u okviru koga se može implementirati veći broj dizajna. Postoje dve opcije za implementaciju *BHSR*-a: *globalna*-(*G*) i *individualna*–(*P*). Globalna implementacija koristi jedinstveni *BHSR* sa *k* ulaza koji prate smerove grananja zadnjih *k* dinamičkih *Branch* instrukcija u programskom izvršenju. To može biti bilo koji broj (*1* do *k*)-statičkih *Branch* instrukcija. Individualne (po *Yeh* i *Patt*-u nazvane *per-branch*) implementacije koriste skup *k*-bitnih *BHSR*-ova (vidi sliku 4.11), pri čemu se selektuje ona koja je bazirana na adresi grananja. U suštini *globalna BHSR* je deljiva od strane svih statičkih grananja, dok kod *individualnih BHSR*-ova svaki *BHSR* se dodeljuje po jednoj statičkoj *Branch* instrukciji, ili podskupu statičkih *Branch*-ova ako postoji adresna alijaza (preklapanje adresa) kod indeksiranja skupa *BHSR*-ova koristeći adresu *Branch*. Postoje tri opcije za implementaciju *PHT*-a: *globalna*-(*g*), *individualna*-(*p*), ili *deljiva*-(*s*). Globalni *PHT* koristi jedinstvenu tabelu da bi podržao predikciju svih statičkih grananja. Alternativno, moguće je koristiti i *individualne PHT*-ove pri čemu je svaki *PHT* namenjen po jednom statičkom grananju (*p*) ili manjem podskupu statičkih grananja (*s*), ako postoje adresne alijaze kada se vrši indeksiranje u skupu *PHT*-ova koristeći *Branch* adrese. Treća dimenzija, da bi se obavilo projektovanje, uključuje implementaciju algoritma aktuelne predikcije. Kada se istorijsko zasnovani *FSM* koristi radi implementacije algoritma o predikciji, *Yeh* i *Patt* su identifikovali ove šeme kao adaptivne (A).

Sve moguće implementacije dvo-nivovske adaptivne *Branch* predikcije se mogu klasifikovati na osnovu tro-dimenzionalnih dizajn parametra. Data implementacija se označava koristeći tro-slovnu notaciju, tj. *GAs* predstavlja dizajn koji koristi jedinstvenu *globalnu BHSR*, adaptivni algoritam za predikciju, i skup *PHT*-ova od kojih je svaka zajednička za veći broj statičkih grananja. *Yeh* i *Patt* su prezentovali tri specifične implemenmtacije koje za skup *Benchmark*-ova postižu tačnost predikcije od 97%. To su sledeće implementacije:

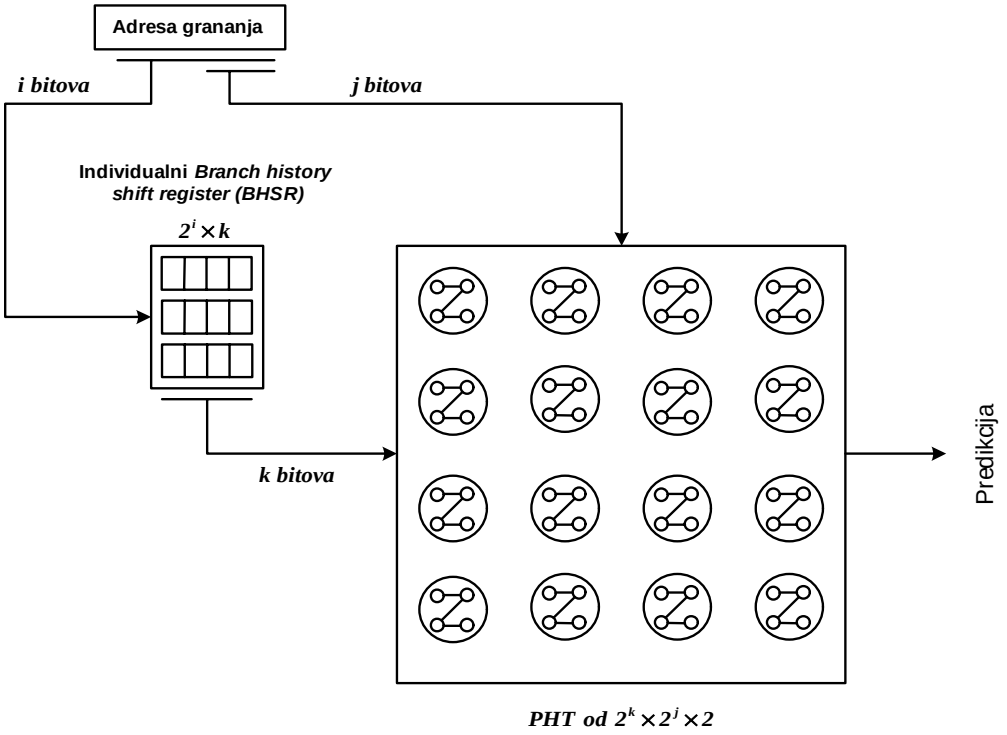
- GAg:(1) *BHSR* obima 18 bitova; (1) *PHT* obima $2^{18} \times 2$ bita
- PAg: (512×4) *BHSR*-ova obima: 12 bitova; (1) *PHT* obima $2^{12} \times 2$ bitova
- PAs: (512×4) *BHSR*-ova obima: 12 bitova; (512) *PHT* obima $2^{6} \times 2$ bitova

Sve tri implementacije koriste adaptivni (A) prediktor koji je dvo-bitna *FSM*. Prva implementacija koristi *globalnu BHSR* (G) sa 18 bitova i *globalnu PHT* (g) sa 2^{18} ulaza koji se indeksiraju od strane *BHSR* bitova. Druga implementacija koristi 512 skupova (4-struki skupno-asocijativni) sa 12-to bitnim *BHSR*-ovima (P) i globalni *PHT* (g) sa 2^{12} ulaza. Treća implementacija takodje koristi 512 skupova 4-struko skupno asocijativne *BHSR*-ove (P) ali svaki obima 6 bitova. Ona takodje koristi 512 *PHT*-ova (s) od kojih svaki ima 2^{6} ulaza indeksiranih od strane *BHSR* bitova. Oba 512 skupova *BHSR*-ova i 512 *PHT*-ova se indeksiraju pomoću 9 bitova *Branch* adrese. Dodatni bitovi *Branch* adrese se koriste za skupno-asocijativni pristup *BHSR*-ovima. *PHT* tabele sa 512 ulaza su direktno preslikani, pa zbog toga može da postoji alijaza tj. veći broj *Branch* adresa da deli isti *PHT*. Na osnovu eksperimentalnih podataka, ovakve alijaze imaju minimalni uticaj na degradaciju tačnosti predikcije. Postizanje veće tačnosti predikcije od 95% od strane dvo-nivovske adaptivne *Branch* predikcione šeme predstavlja zaista impresivni rezultat; najbolje tradicionalne predikcione tehnike mogu da ostvare tačnost predikcije od 90%. Dvo-nivovska adaptivna *Branch* predikcija kao pristup je usvojena od strane većeg broja realnih dizajna, uključujući tu Intel Pentium Pro i AMD /NexGen Nx686.

Na osnovu originalnog *Yeh*-ovog i *Patt*-ovog predloga, *McFarling* (1993), *Young* (1995) i *Gloy* (1996) su poboljšali dvo-nivovsku adaptivnu tehniku, koristeći tzv. korelisane *Branch* prediktore. Na slici 4.12 prikazan je korelisani *Branch* prediktor koji ima *globalnu BHSR* (G) i deljivu *PHT* (s). Dvo-bitni saturacioni brojač se koristi kao *FSM* prediktor. Globalni *BHSR* prati smerove grananja zadnjih *k* dinamičkih grananja i na osnovu toga dinamički kontroliše kontekst toka upravljanja. *PHT* se može posmatrati kao jedinstvena tabela koju čini dvo-dimenzionalno polje sa 2^{j} kolona i 2^{k} vrsta dvo-bitnih prediktora. Ako *Branch* adresa ima *n* bitova, podskup *j* bitova se koristi za indeksiranje *PHT*-a radi selekcije jedne od 2^{j} kolona. S obzirom da je *j* manje od *n*, može doći do odredjenih alijaza pri čemu dve različite *Branch* adrese mogu da indeksiraju istu kolonu *PHT*-a. Zbog toga kažemo da se koristi deljivi *PHT*. Pri tome *k* bitova *BHSR*-a se koriste za selekciju jednog od 2^{k} ulaza selektovane kolone. Dva bita istorije u selektovanom ulazu se koriste za kreiranje predikcije zasnovane na istoriji. Tradicionalna tabela istorije grananja akvivalentna je samo jednoj vrsti *PHT*-a koja se indeksira pomoću *j* bitova *Branch* adrese, kako je to prikazano na slici 4.12 u isprekidano uokvirenom pravougaoniku koga čine dvo-bitni prediktori prve vrste *PHT*-a.



Slika 4.12 Korelisani *Branch* prediktor sa globalnim *BHSR*-ovima i deljivim *PHT*-ovima (GAs)

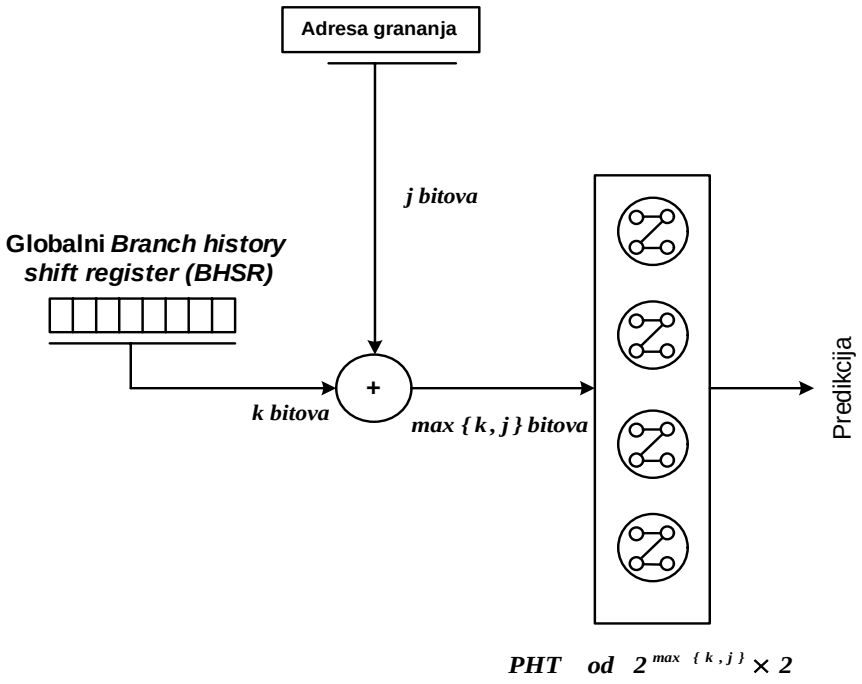


Slika 4.13 Korelisani *Branch* prediktor sa individualnim *BHSR*-ovima i deljivim *PHT*-ovima (PAs)

Slika 4.13 prikazuje korelisani *Branch* prediktor sa individualnim ili *per-branch BHSR*-ovima (P) i istim deljivim *PHT* (s). Slično kao i GAs šema, PAs šema takodje koristi *j* bitova *Branch* adrese

radi selekcije jedne od 2^j kolona *PHT*-a. Ipak, *i* bitova adrese *Branch*, koji se mogu preklapati sa *j* bitova koji su namenjeni za pristup *PHT*-u, koriste se za indeksiranje skupa *BHSR*-ova. U zavisnosti od *Branch* adrese bira se jedna od 2^i *BHSR*-ova. Zbog toga svakoj *BHSR* se pridružuje jedna *Branch* adresa, ili skup *Branch* adresa za slučaj da postoje alijaze. U suštini, umesto da se koristi jedinstvena *BHSR* koja obezbedjuje dinamičko upravljanje kontekstom programa za sva statička grananja, koristi se veći broj *BHSR*-ova koji obezbedjuju različite kontekste dinamičkog upravljanja tokom programa za različite podskupove statičkih grananja. Ovo omogućava dodatnu fleksibilnost praćenja i korišćenja različitih korelacija koje postoje medju različitih instrukcija grananja. Svaki *BHSR* prati smerove zadnjih *k* dinamičkih grananja koje pripadaju istom podskupu statičkih grananja. Kako GAs tako i PAs šeme zahtevaju *PHT* obima $2^k \times 2^j \times 2$ bita. GAs šema ima jedan *k*-to bitni *BHSR*, dok PAs šema zahteva 2^i *k*-to bitnih *BHSR*-ova.

Veoma efikasni korelisani *Branch* prediktor nazvan *gshare* predložen od strane *Scott McFarling* (1993). Kod ove šeme *j* bitova *Branch* adrese su ‘’*hashed*’’ pomoću po-bitne XOR funkcije sa *k* bitova iz globalne *BHSR* (vidi sliku 4.14). Rezultantni $\max\{k, j\}$ bitova koriste se da indeksiranje *PHT*-a obima $2^{\max\{k,j\}}$ bitova radi selekcije jednog od $2^{\max\{k,j\}}$ dvo-bitnih *Branch* prediktora. Kod šeme *gshare* potreban je samo jadan *k*-to bitni *BHSR* i veći broj manjih *PHT*-ova pri čemu se ostvaruju komparativne tačnosti u predikciji u odnosu na druge korelisane *Branch* prediktore. Ova šema se koristi kod Dec Alpha 21264 četvoro-strukog superskalarnog mikroprocesora.



Slika 4.14 *McFarling*-ov *gshare* korelisani *Branch* prediktor

4.1.6. Druge tehnike za upravljanje tokom programa

Primarni cilj tehnika za upravljanje tokom programa je da dostavi što je moguće veći broj korisnih instrukcija izvršnom jezgru procesora u toku svakog mašinskog ciklusa. Dva glavna izazova se odnose na uslovna grananja i slučaj kada *do-grananja-dolazi*. Superskalarni procesori velikog obima, da bi obezbedili adekvatnu propusnost kod instrukcija uslovnog grananja, moraju da ostvare tačnu

predikciju ishoda i cilja grananja za veći broj uslovnih grananja u toku svakog mašinskog ciklusa. Na primer, kod *fetch* grupe sa četiri instrukcije, moguće je da sve četiri instrukcije budu tipa uslovnog grananja. U idealnom slučaju poželjno je da se koriste adrese za sve četiri instrukcije radi indeksiranja četvoro-portnog *BTB*-a kako bi se izvukli bitovi istorije ciljnih adresa za sva četiri grananja. Jedan kompleksni prediktor može zatim da učini jednu celovitu predikciju na osnovu svih bitova istorije. Nakon toga može da sledi spekulativno pribavljanje zasnovano na ovoj predikciji. Tehnike za predikciju kod višestrukih grananja u svakom ciklusu predložene su od strane *Conte* (1995) i *Rotenberg* (1996). Veoma je važno da se obezbedi visoka tačnost kod ovakvih predikcija. U ovom slučaju koristi se istorija o globalnom grananju zajedno sa *per-branch* istorijom sa ciljem da se postigne visoka tačnost predikcije. Za ona grananja ili sekvencu grananja koja ne poseduju strogo naglešeno *Branch* ponašanje veoma je teško izvršiti tačnu predikciju pa se zbog toga koristi *dynamic eager execution – DEE* tehnika predložena od strane *Gus Uht* (1995). *DEE* koristi veći broj PC-ova za simultano pribavljanje većeg broja adresa. U suštini stepen *Fetch* prosledjuje veći broj puteva upravljanja tokom programa sve dok se ishod grananja ne razreši, pa se u tom trenutku na neki od pogrešnih puteva invalidiraju instrukcije.

Slučaj kada *do-grananja-dolazi* je druga glavna prepreka za dostavljanje dovoljnog broja korisnih instrukcija *EX* jezgru. Kod mašina velikog obima jedinica *Fetch* mora biti u stanju da korektno procesira više od jedne instrukcije kod koje *do-grananja-dolazi* u jednom ciklusu, što podrazumeva predikciju svakog smera kao i cilja grananja, kao i pribavljanje, poravnanje i kobinovanje instrukcija sa većeg broja ciljnih adresa grananja. Efektivni pristup za rešavanje ovog problema zasniva se na korošćenju *trace cache* koji je predložen od strane *Rotenberg*-a (1996). Nakon ovoga odredjeni oblik *trace cache*-a je implementiran kod najnovijih verzija Pentium 4 superskalrnog procesora. *Trace cache* je istorijski zasnovan mehanizam pribavljanja i memorisanja tragova dinamičkih instrukcija u kešu koji se indeksira od strane *fetch* adrese i *Branch* ishoda. Ovi tragovi se dinamički asembliraju shodno dinamičkom ponašanju grananja i mogu da sadrže veći broj neredoslednih osnovnih blokova. Kad se javi pogodak kod pribavljanja adresa u *trace cache*-u, instrukcije se pribavljaju iz *trace cache*-a, a ne iz instrukcionog keša. S obzirom da dinamička sekvenca instrukcija u *trace cache*-u može da sadrži veći broj da *do-grananja-dolazi*, a ipak je memorisana sekvencijalno, ne postoji potreba za pribavljanje većeg broja ciljnih adresa grananja, kao i potreba za multi-portnim instrukcionim kešom i logikom kompleksnog mešanja i poravnanja u *Fetch* stepenu. *Trace cache* se može posmatrati kao blok koji obavlja dinamičko preuredjenje u saglasnosti sa dominatnim putevima izvršenja u programu. Mešanje i poravnjanje se obavlja u stepenu *completion*, kada se prvo izvršavaju neredosledni blokovi, koji se nalaze na dominatnom putu, zatim se asemblira informacija o trasi koja se zatim memoriše u jednu liniju *trace cache*-a. Cilj je da nakon što se *trace cache* aktivira, najveći broj pribavljenja će dolaziti sa *trace cache*-a, a ne sa instrukcionog keša. S obzirom da preuredjeni osnovni blokovi u *trace cache*-u mogu znatno bolje da prate dinamički redosled izvršenja, postoji manji broj pribavljanja sa neredoslednih lokacija *trace cache*-a, a shodno tome dolazi do povećanja efektivne propusnosti insrtrukcija kod kojih *do-grananja-dolazi*.

4.2. Tehnike za registarski-protok-podataka

Tehnike za registarski protok podataka odnose se na efektivno izvršenje *ALU*-ih (ili regisarsko-registarskih) tipova instrukcija u izvršnom delu (jezgru) procesora. U suštini može se smatrati da *ALU* instrukcije obavljaju "realni" posao specificiran od strane programa, dok im upravljačke i *Load/Store* instrukcije pružaju ulogu podrške u obezbedjivanju kako neophodnih instrukcija tako i zahtevanih podataka, respektivno. Kod najvećeg broja idealnih mašina, instrukcije tipa *Branch* i *Load/Store*, predstavljaju *overhead* instrukcije, vreme njihovog procesiranja kroz stepen *EX* je nula, ali je latencija izračunavanja, odredjena od strane procesiranja instrukcija tipa *ALU*, najkraće (vreme procesiranja instrukcija kroz *EX* stepen je najkraće kada se izvršava operacija tipa *ALU integer*). Efektivno vreme procesiranja ovih instrukcija predstavlja osnovu za postizanje visokih-performansi.

Ako je naša arhitektura tipa *Load/Store*, tada se *ALU* instrukcijama specificiraju operacije koje se obavljaju nad izvornim operandima koji se čuvaju u registrima. Obično *ALU* instrukcija specificira binarnu operaciju, dva registra iz kojih se izbavljaju operandi, i odredišni registar gde se specificira kao $R_i \leftarrow F_n(R_j, R_k)$ za čije je izvršenje potrebno:

a) funkcionalna jedinica F_n ,
b) dva izvorišna registra R_j i R_k ,
c) odredišni registar R_i

U slučaju da funkcinalna jedinica F_n ne bude dostupna tada kažemo da postoji strukturna zavisnost, a to dovodi do strukturne zavisnosti. Ako jedan ili oba izvorna operanda R_j i R_k nisu dostupna, tada može da dodje (ili dolazi) do pojave hazarda tipa *prave-zavisnosti-po-podacima*. Ako odredišni registar R_i nije dostupan tada se javlja hazard tipa *anti-* ili *izlazna–zavisnost*, tj. tipa lažnih zavisnosti.

4.2.1. Ponovno korišćenje registara i lažnih *zavisnosti po podacima*

Uzrok pojave lažnih zavisnosti predstavlja ponovno korišćenje registara. Ako se neki od registara ne koristi iznova za potrebe memorisanja operanda, tada kažemo da do pojave lažne zavisnosati po podacima i ne dolazi. Ponovno korišćenje registara se naziva *recikliranje registara* (*register recycling*). Postoje dve forme recikliranja registra, statička i dinamička. Statička forma se javlja zbog optimizacije koja se obavlja od strane kompilatora, i nju ćemo prvo sagledati. Kod tipičnog kompajlera, u toku "*back-end*" procesa kompilacije obavljaju se sledeća dva zadatka:

- generisanje kôda (*code generation*)
- dodela registara (*register allocation*)

Proces generisanja kôda je zadužen za aktuelnu emisiju mašinskih instrukcija. Obično generator kôda usvaja da ima na raspolaganju neograničen broj simboličkih registara u kojima smešta sve privremene podatke. Svaki simbolički registar se koristi za memorisanje po jedne vrednosti i u njemu se upisuje samo po jedanput, a ovakav kôd nazivamo *kôd jednostruke-dodele* (*single-assignement code*). Ipak realna ISA ima ograničen broj arhitekturnih (ugradjenih) registara, pa se zbog toga koriste sredstva za dodelu registara koji preslikavaju neograničen broj simboličkih registara u ograničeni i fiksni broj arhitekturnih registara. *Registar-alokator* (deo kompajlera koji vrši dodelu registara) pokušava da zadrži (očuva) što je moguće veći broj privremeno promenljivih u registrima i da pri tome izbegne premeštanje podataka iz registara u memoriju, a zatim ponovo punjenje u registre ako ti podaci zatrebaju. Ovaj zadatak se obavlja ponovnim korišćenjem registara. U registar se upisuje nova vrednost kada stara vrednost koja se čuva u njemu nije više potrebna, a to znači da se sadržaj svakog registra može reciklirati kako bi se čuvao veći broj vrednocti.

Upis u registar se naziva ***definicija* registara,** a čitanje registara nazivamo *korišćenje*. Nakon svake definicije može da postoji jedan ili veći broj korišćenja. Vremenski period izmedju definicije i

zadnjeg korišćenja vrednosti naziva se *životni-vek* (*live range*) te vrednosti. Nakon zadnjeg korišćenja u životnom-veku, registar se može koristiti za memorisanje druge vrednosti pri čemu počinje novi životni-vek. Procedure za alokaciju registara pokušavaju da preslikaju nepreklapajuće životne vekove u iste arhitekturne registre, a to znači da maksimiziraju višestruko korišćenje registara. Kod programa koji vrši jednostavnu dodelu (*single assignmement code*) izmedju simboličkih registara i vrednosti postoji korespondencija *jedan-na-prema-jedan*.

Kada se instrukcije izvršavaju sekvencijalno, a redefiniciji nije nikad dozvoljeno da se dogodi pre prethodne definicije ili zadnjeg korišćenja prethodne definicije, tada životni-vekovi, koji dele (koriste) isti registar, neće nikad uzrokovati probleme. Efektivno, korespondencija *jedan-na-prema-jedan* izmedju vrednosti i registara se može implicitno očuvati ako se sve instrukcije procesiraju u originalnom programskom redosledu. Ipak kod superskalarnih mašina, posebno kada postoji van-redosledno procesiranje instrukcija, operacije čitanja i upis u registre se obavlja u redosledu koji je različit od programskog redosleda. Saglasno tome, korespondencija tipa *jedan-na-prema-jedan* izmedju vrednosti i registara može biti narušena. Zbog toga, sa ciljem da se očuva semantička korektnost sve ***anti-* i *izlazne-zavisnosti*** moraju biti detektovane i registrovane. *Van-redosledno* čitanje (upis u) registre se može dozvoliti sve dok su anti- (izlazne) zavisnosti registrovane.

Dinamički oblik registarskog recikliranja javlja se kada se repetitivno izvršavaju instrukcije u petlji. Agresivna superskalarna mašina koja je u stanju da "u letu" podrži izvršenje većeg broja instrukcija, a pri tome izvršava petlju koja ima relativno malo telo, dozvoljava da se veći broj iteracija petlje mogu istovremeno naći u mašini. To znači da veći broj kopija registra, a koje pripadaju instrukcijama iz različitih operacija, može istovremeno da bude prisutno u mašini, što uzrokuje jednu dinamičku formu registarskog recikliranja. Saglasno prethodnom izmedju dinamičkih instrukcija koje pripadaju višestrukim iteracijama u petlji javljaju se *anti-* i *izlazne-zavisnosti*, pa se zbog toga one moraju detektovati i zapamtiti kako bi se očuvala semantička korektnost izvršenja programa.

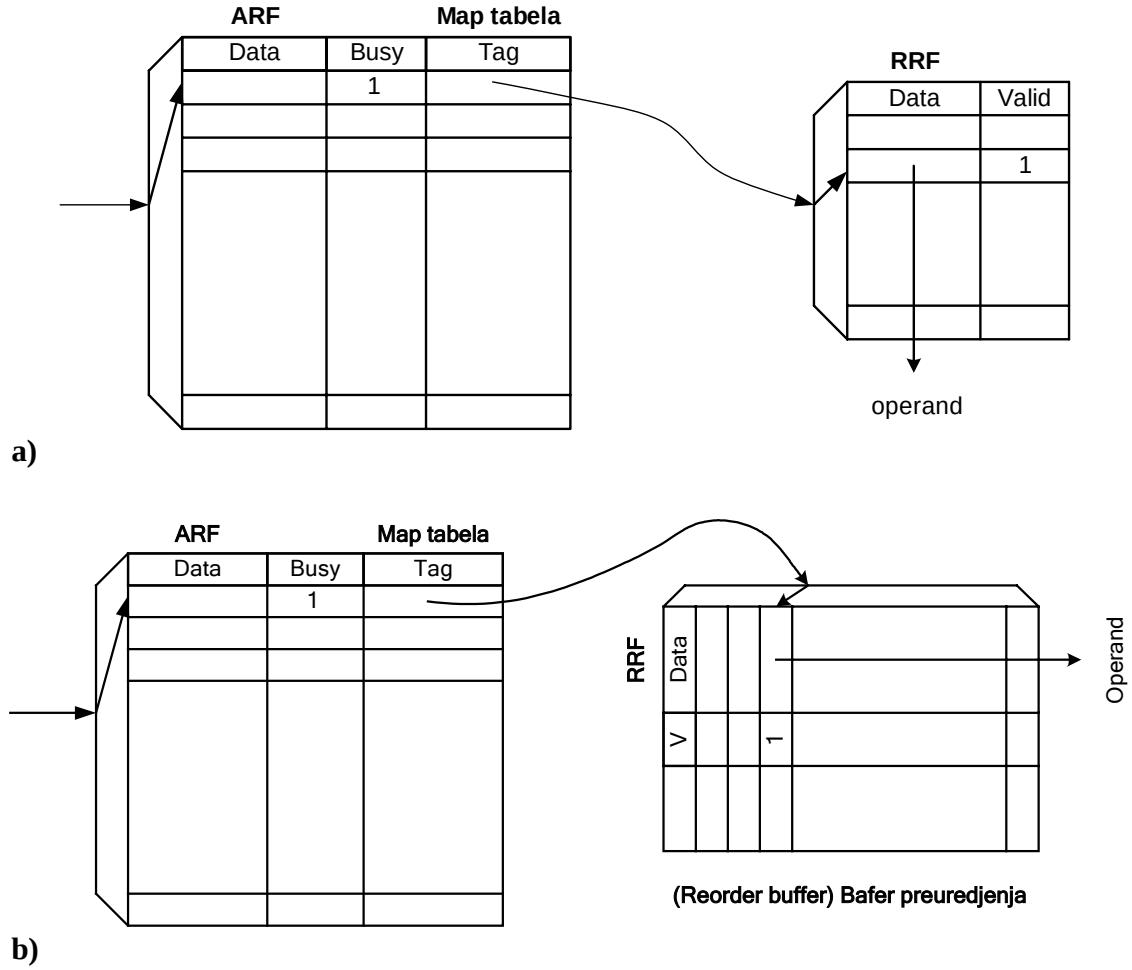
Jedan od načina da se izadje na kraj sa *anti-* i *izlaznim-zavisnostima* sastoji se u zaustavljanju napredovanja zavisne instrukcije sve dok vodeća instrukcija ne završi sa pristupom koji se odnosi na zavisni registar. Tako na primer, deo izmedju jednog para instrukcija postoji *anti* (*WAR*)-zavisnost, tada prateća instrukcija (instrukcija koja ažurira registar) mora da se zaustavi sve dok vodeća instrukcija ne pročita zavisni registar. Sa druge strane, ako izmedju jednog para instrukcija postoji *izlazna* (*WAW*)-*zavisnost*, tada prateća instrukcija (instrukcija koja ažurira registar) mora da se uzaustavi sve dok prvo vodeća instrukcija ne ažurira registar. Ovakvo zaustavljanje u napredovanju *anti-* i *izlazno-zavisnih* instrukcija može dovesti do značajne degradacije performansi i nije neophodno. Naglasimo da su ovakve lažne *zavisnosti-po-podacima* uzrokovane recikliranjem arhitekturnih registara, a nisu uradjene semantikom programa.

4.2.2. Tehnike za preimenovanje registara

Znatno agresivniji način da se izadje na kraj sa lažnim (false) zavisnostima-po-podacima, sastoji se u dinamičkoj dodeli različitih imena, putem višestrukih definicija, jednom te istom arhitekturnom registru. Kao rezultat ostvaruje se eliminisanje postojanja lažnih zavisnosti. Ovu tehniku nazivamo preimenovanje registra (*register renaming*). Za sve instrukcije koje se mogu tekuće simultano izvršavati, ovakav pristup, u toku izvršenja programa, iziskuje poništavanje efekta registarskog recikliranja stvarajući pri tome odnos *jedan-na-prema-jedan* tj. jedan registar jedna vrednost izmedju registara i vrednosti. Za svaku od instrukcija koja se izvršava, ova tehnika sprovodi jedinstvenu dodelu. Pri tome treba naglasiti, da izmedju ovih instrukcija, i dalje postoje *anti-* i *izlazne-zavisnosti*. U suštini ovakvim pristupom se obezbedjuje paralelno izvršenje originalnih instrukcija, i pored toga što izmedju njih i dalje postoje zavisnosti.

Standardni način za implementaciju tehnike preimenovanja registara sastoji se u korišćenju posebnog (dodatnog) registarskog polja za preimenovanje (*Rename Register File- RRF*) pored (već

standardnog) arhitekturnog registarskog polja (*Architected Register File-ARF*). Jedan standardni pristup za implementaciju RRF-a se sastoji u dupliciranju ARF-a i korišćenju RRF-a kao "rezervna" verzija ARF-a. Ovo omogućava da svaki arhitekturni registar bude preimenovan samo jedanput. Ipak ovo rešenje predstavlja efikasan način za korišćenje registara RRF-a Veći broj rešenja implementira RRF sa manjim brojem ulaza u odnosu na ARF, ali omogućava da se svaki od registara RRF polja fleksibilno koristi kod preimenovanja u tom smislu što se svaki registar RRF polja može preimenovati u bilo koji drugi registar iz grupe arhitekturnih registara, tj. registar iz ARF polja. Ovaj način olakšava efikasno korišćenje tehnike preimenovanja registara, ali zahteva ugradnju tabele preslikavanja u kojoj se pamte pokazivači na ulaz u RRF. Princip korišćenja RRF-a u zajedništvu sa tabelom preslikavanja koja obavlja preimenovanje ARF-a prikazana je na slici 4.15.



Slika 4.15 Implementacije RRF-a :(a) samostalna ; (b) pridružena baferu preuredjenja

Napomena:: ***ARF***- (*architected register file*)-arhitekturno registarsko polje**; *RRF*-** (*rename register file*)- registarsko polje za potrebe preimenovanja

Kada se za preimenovanje registra koristi posebno RRF, tada postoje implementacioni izbori koji se razlikuju po tome gde treba smestiti RRF. Jedna opcija kako je to prikazano na slici 4.15 a) se sastoji u implementaciji izdvojene samostalne strukture koja je slična ARF-u a locira se u blizini ARF-a. Alternativno rešenje (vidi sliku 4.15 b) se sastoji u inkorporaciji RRF-a kao deo bafera preuredjenja (*reorder buffer*). Kod obe opcije ARF-u su pridružena dva polja "*busy*" (zauzet) i "*Map table*" (tabela preslikavanja). Ako je "*busy*" bit selektovanog ulaza ARF-a zauzet, to znači da je arhitekturni registar

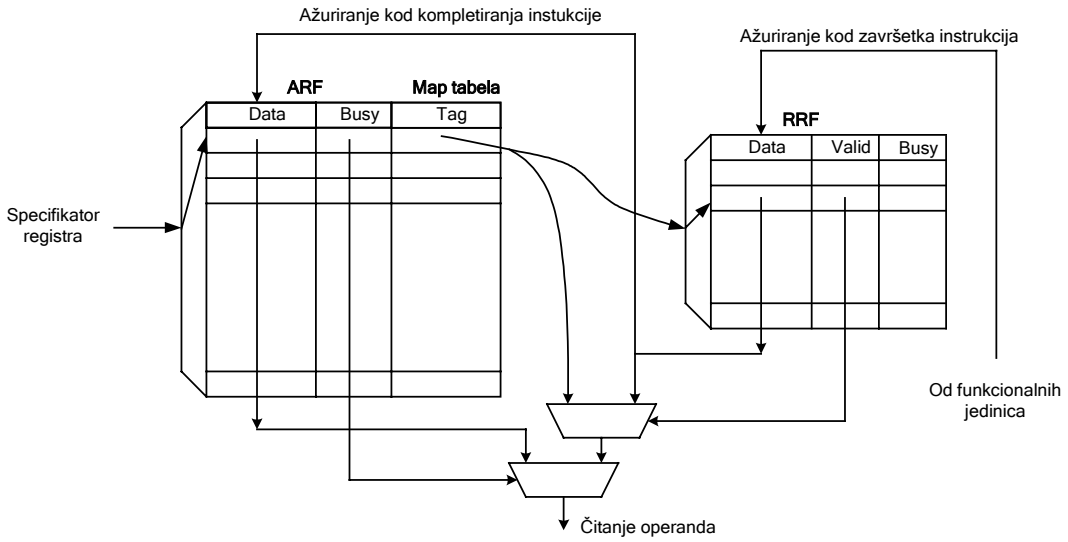
preimenovan, pa se tada odgovarajućem ulazu u "*Map tabeli*" pristupa da bi se pribavio "*tag*" (adresni marker) ili pokazivač na ulaz u RRF. Kod prve opcije, *tag* specificira primenovani registar i koristi se kao indeks u RRF, dok kod druge opcije, *tag* specificira ulaz u bafer preuredjenja i koristi se kao indeks u baferu preuredjenja.

Na osnovu dijagrama sa slike 4.15, može se zaključiti razlika izmedju pomenute dve opcije može izgledati veštačka. No ipak treba ukazati da postoje ključne suptilne razlike. Kao prvo, ako se RRF inkorporira kao deo bafera preuredjenja, tada svaki ulaz bafera preuredjenja imaće dodatno polje koje funkcioniše kao preimenovani registar, pa shodno tome postojaće i preimenovani registar koji se dodeljuje svakoj instrukciji koja se tekuće izvršava. Ovakav dizajn se bazira na scenariju najgoreg slučaja i može izgledati "rasipnički" ako se ima u vidu da se svakom instrukcijom ne definiše registar. Na primer, instrukcije tipa *Branch* ne ažuriraju bilo koji arhitekturni registar. Sa druge strane, bafer preuredjenja već ima ugradjeno portove preko kojih se vrši prihvatanje podataka od funkcionalnih jedinica a takodje i ažuriranje ARF-a, (ove aktivnosti se obavljaju u fazi protočne obrade koja se naziva "*completion*" instrukcije). Kada se ugradi posebno (samostalno) RRF, tada RRF polje treba da sadrži dodatnu strukturu kojoj su potrebni portovi da bi se prihvatili podaci kako iz funkcionalnih jedinica tako podaci koji se odnose na ažuriranje ARF-a. Izbor koji se odnosi na osnovu projektantskih kompromisa. U principu, u praksi, sreću se oba rešenja. Ukazaćemo sada prvo na samostalnu opciju sa ciljem da bi saznali detalje koji se odnose na to kako stvarno funkcioniše tehnika o preimenovanju registra.

Ativnost koja je u vezi sa preimenovanjem registra čine sledećea tri zadatka:

(1) čitanje izvorišta;
(2) dodela odredišta; i
(3) ažuriranje registra.

Prvi zadatak koji se odnosi na čitanje izvorišta obično se dešava u stepenu *Decode* (ili po mogućnosti u stepenu *Dispatch*), a odnosi se na pribavljanje registarskih operanada. U trenutku kada se instrukcija dekodira, njeni izvorišni registarski specifikatori se koriste za indeksiranje multiportnog ARF-a sa ciljem da se pribave registarrski operandi. Tri mogućnosti mogu postojati kod pribavljanja svakog registarskog operanda. Prvo, kada "*Busy*" bit nije postavljen, to znači da ne postoje zahtevi za upis u specificirani registar koji su izdati a nisu usluženi (*pending write to the specified register*) i da arhitekturni registar sadrži specificirani operand, tj. operand se pribavlja iz ARF-a. Alternativno ako je *Busy* bit postavljen, to znači da je već izdat zahtev za upis u taj registar i da je sadržaj arhitekturnog registra zastareo, pa se pristupa odgovarajućem ulazu "*Map*" tabele sa ciljem da se pribavi vrednost *tag*-a kojom se definiše preimenovanje. "*Tag*"-om za preimenovanje specificira se preimenovani registar na taj način što se taj "*tag*" koristi kao indeks u RRF. Preostale dve mogućnosti se mogu sada javiti prilikom indeksiranja RRF-a. Prvo, (druga mogućnost) ako je bit *Valid* indeksiranog ulaza postavljen, to znači da instrukcija koja ažurira registar je već završila sa izvršenjem, ali čeka da bude kompletirana. U tom slučaju, izvorni operand je dostupan u preimenovanom registru i izvalači se (*retreive*) iz indeksiranog RRF ulaza. Kada *Valid* bit nije postavljen, to znači da instrukcija koja ažurira registar nije završena i da preimenovani registar ima zahtev za ažuriranjem koji je aktivan, ali još nije uslužen. U ovom slučaju (treća mogućnost) *tag*, ili specifikator preimenovanog registra, kao izvorni operand se prosledjuje iz "*Map*" tabele ka rezervacionoj stanici. Ovaj *tag* biće kasnije iskorišćen od strane rezervacione stanice da bi se dobio operand kada on postane dostupan. Sve tri mogućnosti koje se odnose na čitanje izvornog operanda prikazane su na slici 4.16.



Slika 4.16 Zadaci koji se odnose na preimenovanje registara: čitanje izvorišnog operanda, dodela odredišta i ažuriranje registara

Zadatak koji se odnosi na alokaciju odredišta se takodje javlja u stepenu *Decode* (a po mogućnosti i u *Dispatch*), i prate ga sledeća tri podzadatka:

a) postavljanje bita *busy* (*set busy bit*);
b) dodela *tag*-a (*assign tag*)
c) ažuriranje *map* tabele (*update map table*).

Kada se instrukcija dekodira, njen odredišni registarski specifikator se koristi za indeksiranje ARF-a. Selektovani arhitekturni registar prati zahtev za upis koji je izdat a nije uslužen (*pending write*), pa se njegov *busy* bit mora setovati. Zbog toga specificirani odredišni registar mora biti preslikan u preimenovani registar. Pri tome se mora selektovati (na osnovu stanja *busy* bita) poseban neiskorišćeni registar koji se preimenjuje. Bit *busy* selektovanog RRF ulaza mora se postaviti, a indeks selektovanog RRF ulaza da se iskoristi kao *tag*. Ovaj *tag* mora biti upisan u odgovarajući ulaz "*map*" tabele, koji će se od strane narednih zavisnih instrukcija koristiti za pribavljanje (njihovih) izvornih operanada.

Zadatak ažuriranja registra se deševa u "*back end*" delu mašine i nije deo aktivnosti aktuelnog preimenovanja registra stepena *Decode/Dispatch*, pa zbog toga nema direktni uticaj na rad RRF-a. Shodno slici 4.16 ažuriranje registra se javlja u dva izdvojena stepena. Kada instrukcija koja ažurira registar završi *Execution* njen rezultat se upisuje u ulaz RRF-a na koji ukazuje *tag*. Kasnije kada je ova instrukcija prešla stepen *Complete* njen rezultat se kopira iz RRF-a u ARF. To znači da ažuriranje registra uključuje prvo ažuriranje ulaza u RRF a zatim ulaza u ARF. Ova dva koraka se javljaju u *jedan-do-drugi* (*back-to-back*) ciklusima u slučaju kada se instrukcija za ažuriranje-registra nalazi na zaglavlju bafera preuredjenja, ili one mogu biti razdvojene većim brojem ciklusa kada postoje i druge nezavršene instrukcije koje se u baferu preuredjenja nalaze ispred ove instrukcije. Nakon što se preimenoivani registar kopira u odgovarajući arhitekturni registar, njegov "*busy*" bit se resetuje i on se ponovo može koristiti za preimenovanje drugog arhitekturnog registra.

U toku dosadašnje analize smo usvojili da implementacija tehnike za preimenovanje registra zahteva korišćenje dva fizička registarska polja, nazvana ARF i RRF. No ovakvo rešenje u suštini nije potrebno. Naime, arhitekturna registre i preimenovane registre možemo zajednički objediniti i implementirati ih kao jedinstveno fizičko registarsko polje pri čemu je broj ulaza jednak sumi ulaza ARF-a i RRF-a. Kod ovakvog objedinjenog registarskog polja, strogo posmatrano, ne pravi se razlika izmedju arhitekturnih registara i registara za preimenovanje. Svaki fizički registar može se fleksibilno odrediti da bude arhitekturni registar ili preimenovani registar. Nasuprot izdvojenoj ARF i RRF

implementaciji koja mora nakon faze *Completition* instrukcije da obavi fizičko kopiranje rezultata iz RRF u ARF, kod objedinjenog registarskog polja neophodno je samo promeniti naznačenost registra od preimenovanog u arhitekturni. Na ovaj način se štedi na vremenu prenosa potrebnog za prelaz informacije iz RRF-a u ARF. Ključni nedostatak objedinjenog registarskog polja je njegova hardverska kompleksnost. Drugi nedostatak odnosi se na vreme komutacije konteksta, kada stanje mašine mora da se zapamti, podskup registara koji čini arhitekturno stanje mašine moraju eksplicitno da se identifikuje pre nego što počne da se pamti stanje.

Najveći broj savremenih superskalarnih mikroprocesora implementiraju odredjeni oblik preimenovanja registra sa ciljem da se izbegnu zastoji zbog *anti-* i *izlazno registarskih zavisnosti po podacima* prvenstveno uzrokovanih zbog višestrukog korišćenja registara. Obično preimenovanje registra se javlja u trenutku dekodiranja instrukcija, a implementacija preimenovanja postaje veoma složena, posebno kod superskalarnih mašina većeg stepena skalarnosti kod koje se simultano mora obaviti preimenovanje većeg broja registarskih specifikatora za veći broj instrukcija. Moguće je takodje i da se veći broj redefinicija registra javi u okviru jedne grupe istih pribavljenih instrukcija (*fetch group*). Implementacija mehanizma preimenovanja kod superskalarnih mašina većeg stepena superskalarnosti bez da se pri tome ima uticaj na vreme-trajanja mašinskog ciklusa predstavlja realni izazov. Da bi se ostvarile visoke performanse mora da se eliminiše serijalizacija ograničenja uslovljena pogrešnim registarskim zavisnostima po podacima, a to znači da se izlaz mora potražiti u dinamičkom preimenovanju registara.

4.2.3. Prave zavisnosti po podacima i ograničenja u protoku podataka

RAW zavisnost izmedju dve instrukcije naziva se *prava-zavisnost-po-podacima* a posledica je egzistencije odnosa *proizvodjač-potrošač* koji egzistira izmedju ove dve instrukcije. Prateća *potrošač-instrukcija* ne može da dobije (pribavi) svoj izvorni operand sve dok vodeća *proizvodjač-instrukcija* ne generiše rezultat. Prava zavisnost po podacima nalaže serijalizaciju ograničenja koji postoje izmedju dve zavisne instrukcije;vodeća instrukcija mora da završi sa izvršenjem pre nego što prateća instrukcija počne sa izvršenjem. Prave zavisnosti po podacima prvenstveno postoje zbog semantike u programu i obično se predstavljaju pomoću grafa zavsnosti po podacima (*Data Dependece Graph* –DDG, ili *Data Flow Graph* –DFG).

Na slici 4.18 prikazan je programski segment koji se odnosi na FFT implementaciju.

```
w[i+k].ip = z[i].rp + z[m+i].rp ;
w[i+j].rp = e[k+1].rp * (z[i].rp - z[m+1].rp) - e[k+1].ip * (z[i].ip - z[m+i].ip);
```

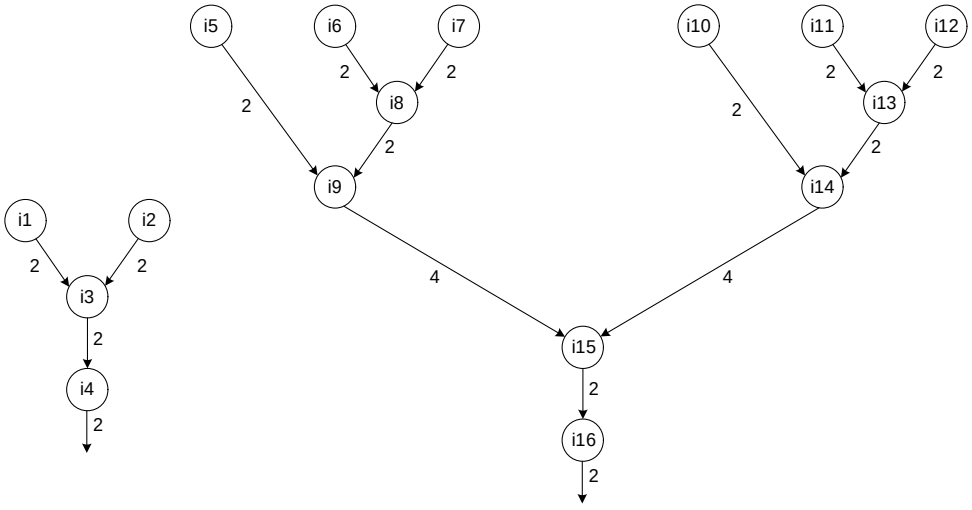
a)

```
i1: f2  ← load,4(r2)
i2: f0  ← load,4(r5)
i3: f0  ← fadd,f2,f0
i4: 4(r6) ← store f0
i5: f14 ←  load,8(r7)
i6: f6  ← load,0(r2)
i7: f5  ← load,0(r3)
i8: f5  ← fsub,f6,f5
i9: f4  ← fmul,f14,f5
i10: f15  ← load,12(r7)
i11: f7  ← load,4(r2)
i12: f8  ← load,4(r3)
i13: f8  ← fsub,f7,f8
i14: f8  ← fmul,f15,f8
i15: f8 ← fsub,f4,f8
i16: 0(r8) ← store,f8
```

b)

Slika 4.18 FFT kôdna sekvenca: (a) originalni izvorni iskazi; (b) kompajlirane asemblerske instrukcije

Kao što se vidi sa slike 4.18 a) dva izvorna iskaza se kompajliraju u 16 aseblerskih instrukcija, uključujući *Load* i *Store* instrukcije. Polje FP promenljivih se čuva u memoriji i mora prvo da se napuni pre nego što obave operacije. Nakon izračunavanja rezultati se ponovo vraćaju u memoriju. *Integer* registri (r_i) se koriste za čuvanje adrese polja. FP registri (f_j) se koriste za čuvanje privremenih podataka. DFG koji odgovara operacijama upis i čitanje FP registara za svih 16 instrukcija sa slike 4.18 b) je prikazan na slici 4.19.



Slika 4.19 DFG za kôdni segement sa slike 4.18 b)

Svaki čvor na slici 4.19 predstavlja po jednu instrukciju na slici 4.18 b). Izmedju dve instrukcije postoji usmereni poteg ako postoji prava zavisnost izmedju te dve instrukcije. Za svaki od potega-zavisnosti u DFG-u može se identifikovati zavisni registar. Svakom od potega-zavisnosti se može dodeliti latencija. Na slici 4.19, svakom potegu je pridružena latentnost izvršenja instrukcije

"proizvodjač". U konkretnom slučaju za instrukcije *Load*, *Store*, *Add*, i *Sub* je usvojeno da imaju dva ciklusa izvršne latentnosti, dok su za instrukcije množenja potrebna 4-ciklusa. Latencije pridružene zavisnim-potezima su kumulativne. Najduži zavisni lanac zavisnosti, meri se u funkciji totalne kumulativne latencije, a identifikuje sa kao kritični put DFG-a. Čak i da usvojimo egzistenciju neograničenog broja mašinskih ciklusa, izvršenje kôdnog segmenta biće direktno (ne može biti brže) a odnosi se na izvršenje programa i predstavlja najbolje performanse koje se ogu postići. Za kôdni fragment sa slike 4.19 DFL iznosi 12 ciklusa. DFL je uslovljeno pravim zavisnostima u programu. Tradicionalno model toka izvršenja (*data flow execution model*) odredjuje da svaka instrukcija u programu počne sa izvršenjem ciklusa koji neposredno sledi od trenutka kada svi njeni operandi postaju dostupni. U suštini, sve postojeće registarske tehnike za protok podataka predstavljaju pokušaji da se reše problemi koji prate ograničenja vezana za tok podataka.

4.2.4. Klasični *Tomasulo* algoritam

Projektovanje FP jedinice (*Floating Point Unit–FPU)* kod računara IBM 360/91, kao i inkorporiranje u tu mašinu *Tomasulo*-vog algoritma, predstavljalo je osnovu za razvoj savremenih superskalarnih procesora. Ključni atributi za najveći broj savremenih tehnika za protok podataka se mogu naći u *Tomasulo* algoritmu, pa zbog toga njegovo izračunavanje zaslužuje posebnu pažnju. Imajući ovo u vidu upoznaćemo se prvo sa polaznim dizajnom *FPU*-a kod IBM 360, zatim opisati modifikovani dizajn *FPU*-a kod IBM 360/91 koja inkorporira *Tomasulo*-ov algoritam, i na kraju ukazati na njenu efikasnost u procesiranju kôdne sekvence.

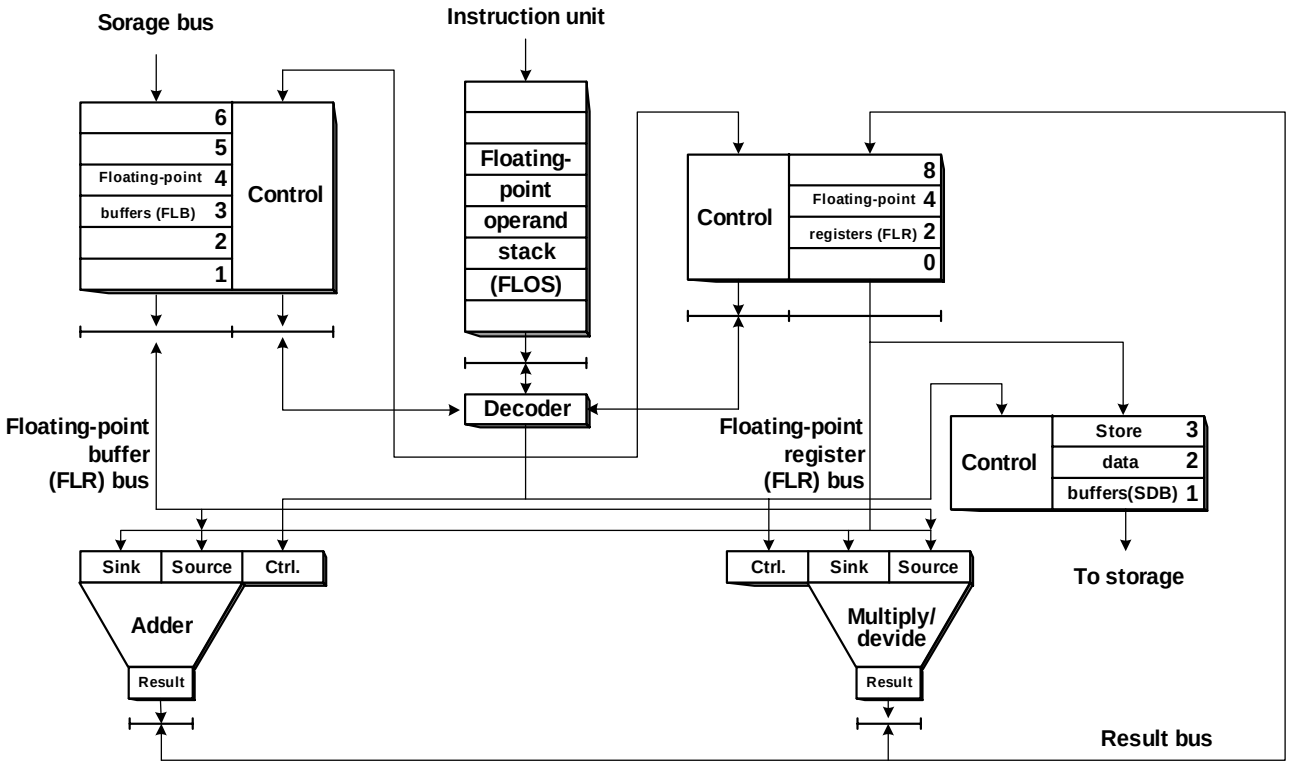
Originalni dizajn *FPU*-a kod IBM 360 je prikazan na slici 4.20. *FPU* sadrži dve izvršne funkcionalne jedinice: jedna tipa *FP-Add*, a druga *FP-Multiply/Devide*. Kod *FPU*-a sa slike 4.10 postoje tri registarska polja: FP registri (FLR), FP baferi (FLB), i baferi za memorisanje podataka (SDB). Postojr četiri FLR registra; to su arhitekturni FLR registri. FP instrukcije koje koriste adresni načini rada tipa memorija-registar ili memorija-memorija se preprocesiraju. Generisanje adrese i pristup memoriji vrši se van *FPU*-a. Kada se podaci dobavljaju iz memorije, oni se pune u jedan od šest FLB-ovih registara. Na sličan način, ako je odredište instrukcije memorijska lokacija, rezultat koji se memoriše treba smestiti u jedan od triju SDB-ovih registara, a zatim pomoću posebne hardverske jedinice treba pristupiti SDB-u koja će rezultat smestiti u memorijsku lokaciju. Korišćenje ova dva dodatna registarska polja, FLB i SDB, radi podrške instrukcijama koje koriste memorija-registar i memorija-memorija adresne načine rada, obezbedjuju se uslovi da *FPU* efikasno funkcioniše kao mašina tipa registar-registar.

Kod IBM 360, instrukciona jedinica (IU) dekodira sve instrukcije i predaje po redosledu (*in order*) sve FP instrukcije ka operativnom FP magacinu (*floating point operation stack – FLOS*). Kod *FPU*-a, FP instrukcije se zatim dekodiraju i iz *FLOS*-a se iniciraju po redosledu radi izvršenja u obe izvršne funkcionalne jedinice (*FP_Add* i *FP_Multiply/Devide*). Izvršne funkcionalne jedinice nisu organizovane na protočni način tako da njihova latentnost iznosi nekoliko ciklusa. Sabiraču je potrebno dva ciklusa da izvrši instrukcije *FADD* dok su jedinici za množenje/deljenje potrebna tri ciklusa *FMUL* i 12 ciklusa za *FDIV*.

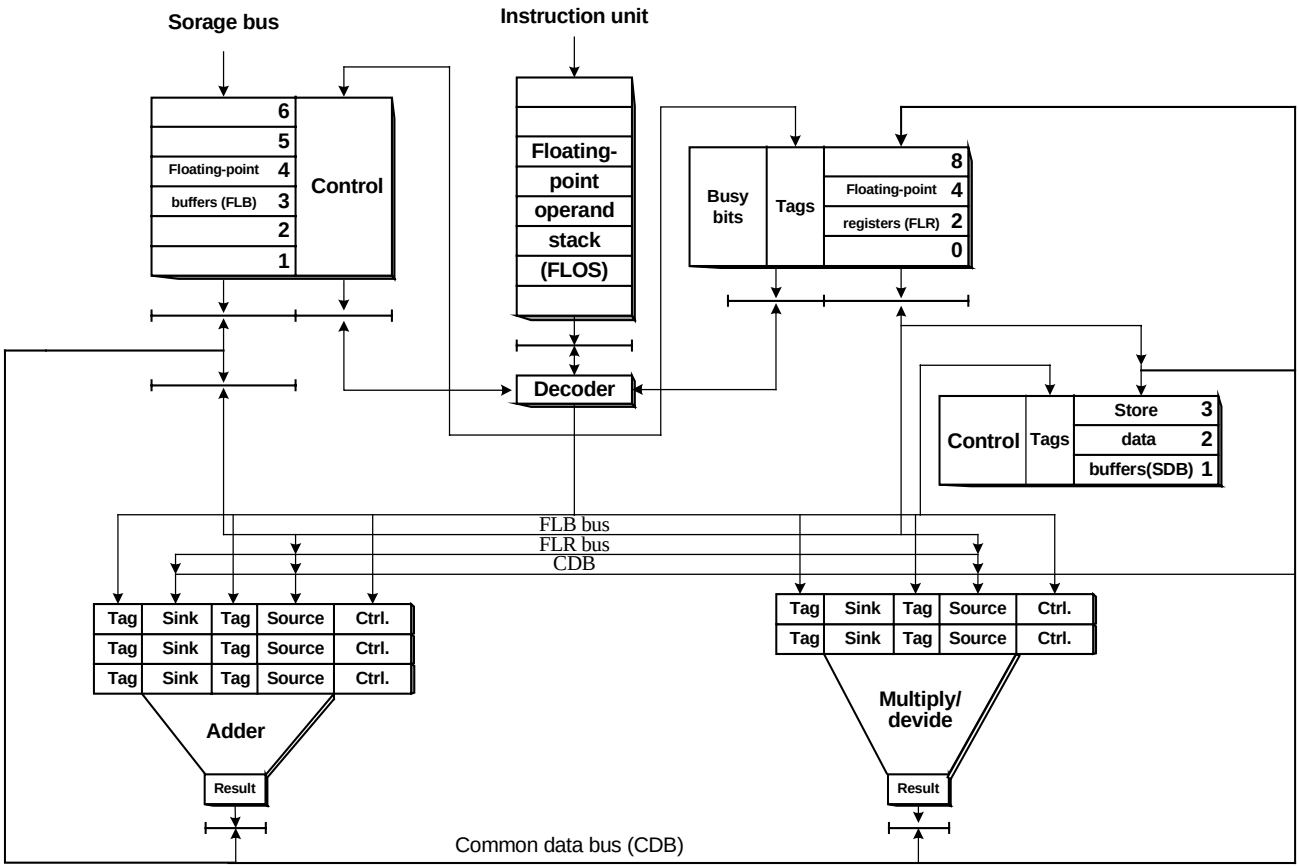
Sredinom šesdesetih godina prošlog veka IBM je počeo sa razvojem Modela 91 koja je kao mašina pripadala familiji 360. Jedan od ciljeva je bio da se postigne konkurentno izvršenje većeg broja FP instrukcija i da se ostvari propusnost od jedne instrukcije po ciklusu kod protočne organizacije u procesiranju instrukcija. Ovakav pristup može izgledati veoma agresivan imajući u vidu:

a) kompleksne načine rada kod ISA 360; i
b) latencija od većeg broja ciklusa u toku izvršenja.

Kao krajnji rezultat se dobija modifikovana *FPU* koja kod 360/91 je inorporirala *Tomasulo*-ov algoritam (vidi sliku 4.21).



Slika 4.20 Originalni dizajn *FPU*-a kod IBM 360



Slika 4.21 Modifikovani dizajn *FPU*-a kod IBM 360/91 koja inkorporira *Tomasulo*-ov algoritam

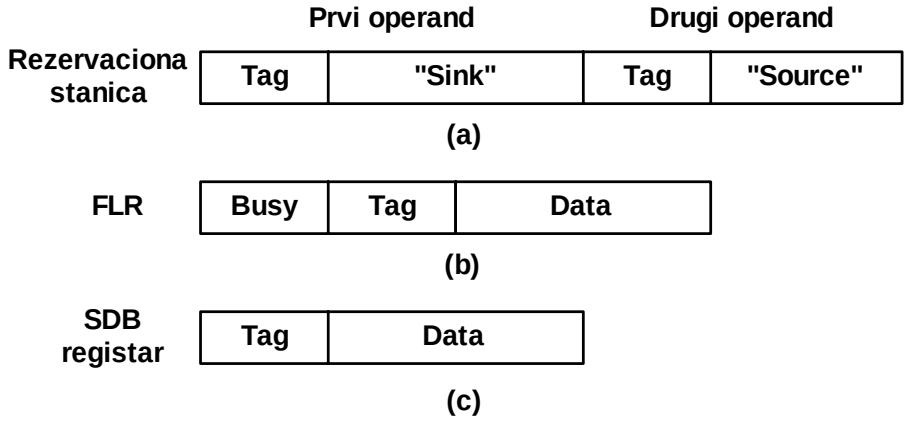
Tomasulo-ov algoritam se bazira na uvodjenu triju novih mehanizama kod originalnog *FPU* dizajna, a to su: rezervacione stanice (*reservation stations*), deljiva magistrala podataka (*common data bus*), i registarski *tag*-ovi (*register tags*). Kod originalne verzije (slika 4.20) svaka izvršna funkcionalana jedinica na stranu ulaza je imala bafer dubine jedan u kome se čuvala instrukcija koja se tekuće izvršava. Kada je funkcionalna jedinica bila zauzeta, iniciranje izvršenja instrukcija od strane *FLOS*-a je bilo zaustavljeno sve dok se tekuća instrukcija ne završi. Da bi se premostilo ovo strukturno usko grlo na ulaznoj strani svake funkcionalne jedinice pridružen je veći broj bafera koje nazivamo *rezervacione stanice*. Jedinica *FP_Add* ima tri dok *FP_Multiply/Devide* ima dva bafera. Rezervacione stanice se mogu posmatrati kao virtuelne funkcionalne jedinice. To znači da sve dok postoji slobodna rezervaciona stanica, *FLOS* može da inicira izvršenje u toj funkcionalnoj jedinici, čak i kada je ta jedinica tekuće zauzeta (*busy*) izvršenjem druge instrukcije. S obzirom da *FLOS* inicira izvršenje instrukcija po redusledu, ovakav pristup će ublažiti (otkloniti) nepotreban zastoj zbog kobnog (neravnomernog) rasporeda (uredjenja) različitih tipova FP instrukcija.

Raspoloživost rezervacionih stanica, omogućava da se od strane *FLOS*-a prema funkcionalnim jedinicama može inicirati izvršenje instrukcije čak i u slučaju kada operandi te instrukcije nisu dostupni. Ova instrukcije mogu u rezervacionoj stanici da čekaju za svoje operande a početi sa izvršenjem kada oni postanu dostupni. Magistrala CDB povezuje izlaze obeju funkcionalnih jedinica sa ulazima rezervacionih stanica kao i sa FLR i SDB registrima. Rezultat generisan od strane funkcionalne jedinice emituje se svima (*broadcast*) preko CDB-a. Sve ove instrukcije, koje se nalaze u rezervacinim stanicama, a potreban im je taj rezultat kao operand, pristupiće se lečovanju podatka koji je trenutno prisutan na CDB-u. Takodje registri FLR-a i SDB-a koji su odredište rezultata takodje će lečovati podatak trenutno prisutan na CDB-u. U suštini CDB obezbedjuje direktno premošćavanje

rezultata od instrukcija tipa proizvodjač do instrukcija tipa potrošač, pri čemu ti rezultati čekaju u rezervacionim stancama bez da prodju preko registara (da se njihove vrednosti dobave iz registara). Ažuriranje odredišnih registara se obavlja simultano zajedno sa premošćavanjem (prosledjivanjem) rezultata ka zavisnim instrukcijama. Ako operand dolazi iz memorijske lokacije, on će se nakon izvršenja memorijskih ciklusa napuniti u FLB registar. Shodno tome, FLB može da postavi svoj izlaz na magistralu CDB, tako da one instrukcije koje čekaju u rezervacionoj stanici mogu da lečuju svoj operand. To znači da obe funkcionalne jedinice i FLB mogu da predaju podatke preko CDB, a da rezervacione stanice, FLR i SDB mogu lečovati podatke prisutne na magistrali CDB.

Kada FLOPS predaje (*dispatch*) instrukciju ka funkcionalnoj jedinici, ona dodeljuje rezervacinu stanicu i proverava kako bi ustanovila da li su potrebni operandi dostupni. Ako je operand dostupan u FLR-u tada sadržaj tog registra se kopira iz FLR-a u rezervacionu stanicu; inače, umesto toga, u rezervacionu stanicu se kopira *tag*. *Tag* (marker) ukazuje da postoji operand na koga se čeka da pristigne (postane dostupan). Operand koji se očekuje (*pending operand*) dolazi iz instrukcije proizvodjač koja se tekuće nalazi u jednoj od pet rezervacionih stanica, ili dolazi sa jednog od šest FLB-ovih registara. Da bi se na jedinstven način identifikovao, jedno od mogućih 11 izvorišta operanda koga očekujemo potrebno je ugraditi 4-bitni *tag*. Ako jedno od oba operandska polja u rezervacionoj stanici sadrži marker, a ne aktuelni oparand, tada to znači da ta instrukcija očekuje operand. Kada operand koga očekujemo postane dostupan, proizvodjač tog operanda preko CDB-a ažurira *tag*, kao i polje u kome se čuva aktuelni operand.

Instrukcija koja u rezervacionoj stanici čeka na raspoloživost svog operanda koristi *tag* da bi nadgledala aktivnosti (monitorisala) na CDB magistrali. Kada instrukcija detektuje *tag* uparivanje sa informacijom prisutnom na CDB magistrali, tada ona lečuje svoj pridruženi operand. U suštini proizvodjač operanada emituje (predaje) *tag* operanda na magistralu CDB, pri čemu svi potrošači tog operanda nadgledaju CDB radi pojavljivanja *tag*-a. Svaka rezervaciona stanica sadrži po dva polja za operande, pri čemu svako polje mora da ima *tag* polje jer svaki od oba operanda može da se očekuje. Sva četiri registra FLR-a kao i tri registra SDB-a moraju imati *tag* polja. To znači da ukupno postoje 17 polja koje nadgledaju (monitorišu) i prihvataju operande (vidi sliku 4.2.2). *Tag* polje na strani svakog potencijalnog potrošača koristi princip asocijativnog pretraživanja u cilju monitorisanja radi mogućeg uparivanja svog sadržaja sa vrednošću *tag*-a koja je emitovana na CDB magistralu. Kada dodje do uparivanja *tag* vrednosti, potrošač lečuje emitovani operand.



Slika 4.22 Korišćene *tag* polja kod (a) rezervacione stanice, (b) FLR registra, (c) SDB registra

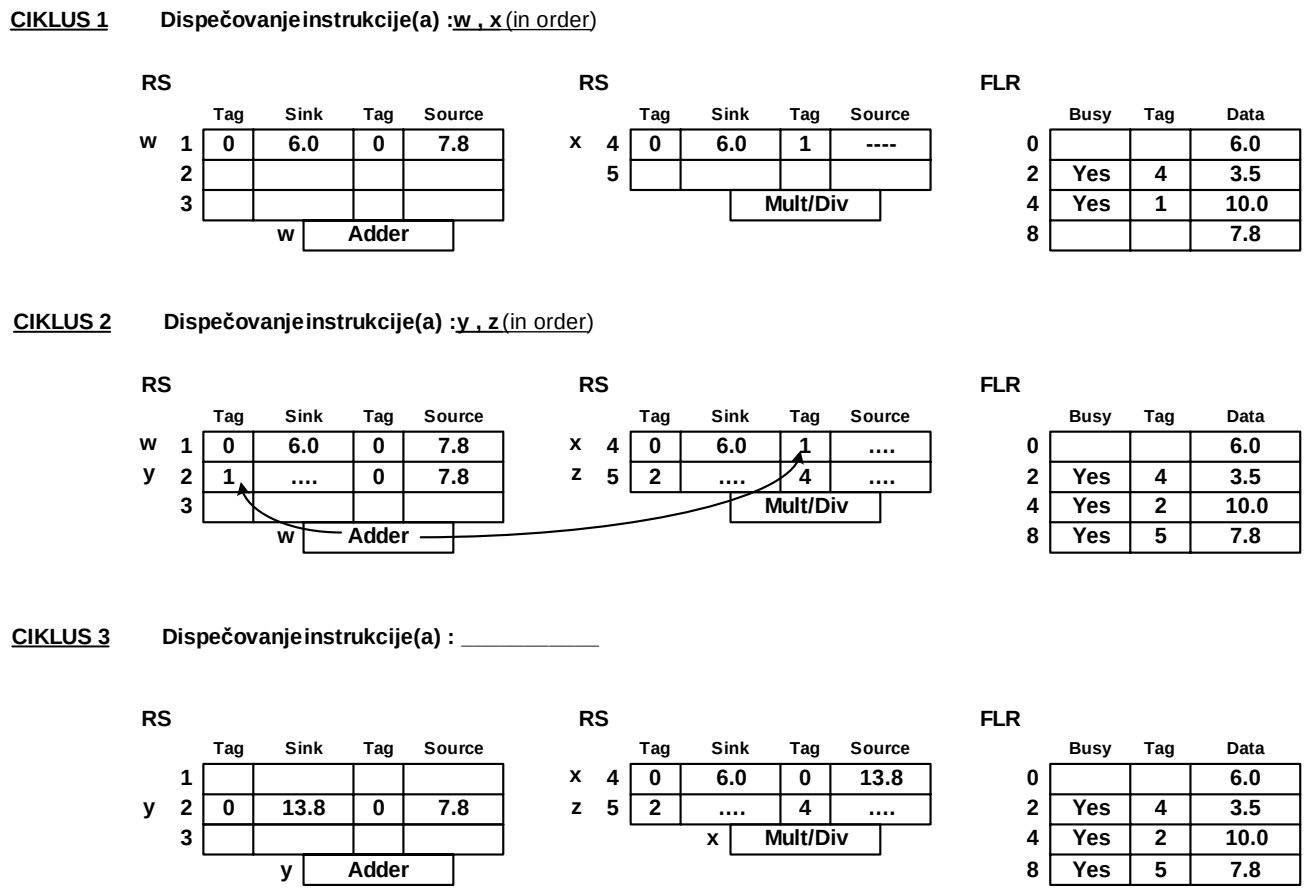
Kod IBM 360 FP instrukcije koriste dvo adresni format, što znači da se specificiraju dva izvorna operanda. Prvi specifikator operanda se naziva *sink* (ponor), a on je istovremeno i odredišni. Drugi operand specifikator zove se *source* (izvorište). Svaka rezervaciona stanica ima dva polja za operande, jedno za *sink* a drugo za *source*. Svakom polju operanda pridružuje se *tag* polje. Ako polje

operanda sadrži realni podatak tada se *tag* polje postavlja na nuli. Inače, njegovo *tag* polje identifikuje *source* odakle operand koji se očekuje treba da dodje (da se dobavi), a koristi se za monitorisanja aktivnosti na CDB magistrali radi dostupnosti na operand koji se očekuje. Kadgod se instrukcija od strane *FLOS*-a dispečuje prema rezervacionoj stanici, podatak koji se čuva u FLR registru, a odgovara *sink* operandu, se izbavlja (izvlači) i kopira u rezervacionu stanicu. Istovremeno postavlja se bit "*busy*" koji se pridružuje FLR registru. Stanje ovog bita ukazuje da predstoji očekivano ažuriranje FLR registra, i da *tag* vrednost koja identifikuje pojedinu rezervacionu stanicu, prema kojoj je instrukcija dispečovana, je upisano u *tag* polje istog FLR registra. Na ovaj način jasno se identifikuje koja će od rezervacionih stanica biti ta koja će generisati ažurirani podatak za ovaj FLR registar. Saglasno tome ako prateća instrukcija specificira ovaj registar kao jedan od njenih izvornih operanada, u trenutku kada se on dispečuje prema rezervacionoj stanici, tada će samo *tag* polje (nazvano pseudo-operand) biti kopirano u odgovarajuće polje rezervacione stanice, a ne i aktuelni podatak. Kada je *busy* postavljen, tada on ukazuje da podatak koji se čuva u FLR registru je zastareo i da vrednost na koju je postavljen *tag* se odnosi na izvorište odakle realni podatak treba očekivati. Osim rezervacionih stanica kao i FLR registara, takodje i SDB registri mogu biti odredišta za operande koje očekujemo (*pending operands*) pa shodno tome svakom od SDB-ovih registara se pridružuje *tag* polje (vidi sliku 4.21).

Na primeru izvršenja jedne sekvence instrukcija ilustrovaćemo rad *Tomasulo*-ovog algoritma. U cilju jasnijeg objašnjenja odstupićemo od opisa kojim se aktuelni IBM 360/91 dizajn razlikuje u nekoliko rešenja. Prvo, da bi se izbegla potencijalna konfuzija, umesto dvo-adresnog formata instrukcija (kakav se koristi kod IBM 360/91) koristićemo, tro-adresne instrukcije. Programsku sekvencu koju ćemo razmotriti sadrži samo instrukcije tipa registar-u-registar. Da bi smanjili broj mašinskih ciklusa koje treba da pratimo dozvolićemo *FLOS*-u da dispečuje (u programskom redosledu) da dve instrukcije u svakom ciklusu. Usvojićemo takodje da instrukcija može da počne sa izvršenjem u istom ciklusu kada je i dispečovana u rezervacionu stanicu. Latencije instrukcija *FAdd* i *FMul* su dva i tri ciklusa, respektivno. Dozvolićemo instrukciji da prosledi svoj rezultat zavisnim instrukcijama u toku zadnjeg ciklusa izvršenja, i da zavisna instrukcija može početi izvršenje u narednom ciklusu. *Tag* vrednosti 1, 2 i 3 se koriste za identifikaciju triju rezervacionih stanica funkcionalne jedinice "*Adder*", dok se 4 i 5 koriste za identifikaciju obeju rezervacionih stanica funkcionalne jedinice "*Multply/Divide*". Ove *tag* vrednosti nazivamo ID-ove rezervacionih stanica. Programsku sekvencu čine sledeće četiri instrukcije tipa *registar-u-registar*.

```
w:   R4 ← R0 + R8
x:   R2 ← R0 * R4
y:   R4 ← R4 * R8
z:   R8 ← R4 * R2
```

Na slici 4-23 prikazana su prva tri ciklusa izvršenja. U ciklusu # 1 dispečuje se (u redosledu) prema rezervacionim stanicama instrukcije *w* i *x* . Odredišni registri instrukcija *w* i *x* su R4 i R2 (tj. FLR-ovi 4 i 2), respektivno. "*Busy*" bitovi ova dva registra se postavljaju. S obzirom da je instrukcija *w* dispečovana (upućena) ka rezervacionoj stanici 1, u *tag* polje za R4 se unosi vrednost 1, a to znači da će instrukcija u rezervacionoj stanici 1 ažurirati R4. Na sličan način *tag* vrednosti 4 se unose u *tag* polje za R2. S obzirom da su oba izvorna operanda dostupna izvršenje instrukcije *w* odmah počinje. Instrukciji *x*, kao drugi (izvorni) operand, potreban je rezultat (R4) koga generiše instrukcija *w*. To znači da kada se instrukcija *x* dispečuje ka rezervacionoj stanici 4, u *tag* polje drugog operanda se unosi *tag* vrednost 1, što ukazuje da će instrukcija u rezervacionoj stanici 1 generisati potreban operand.

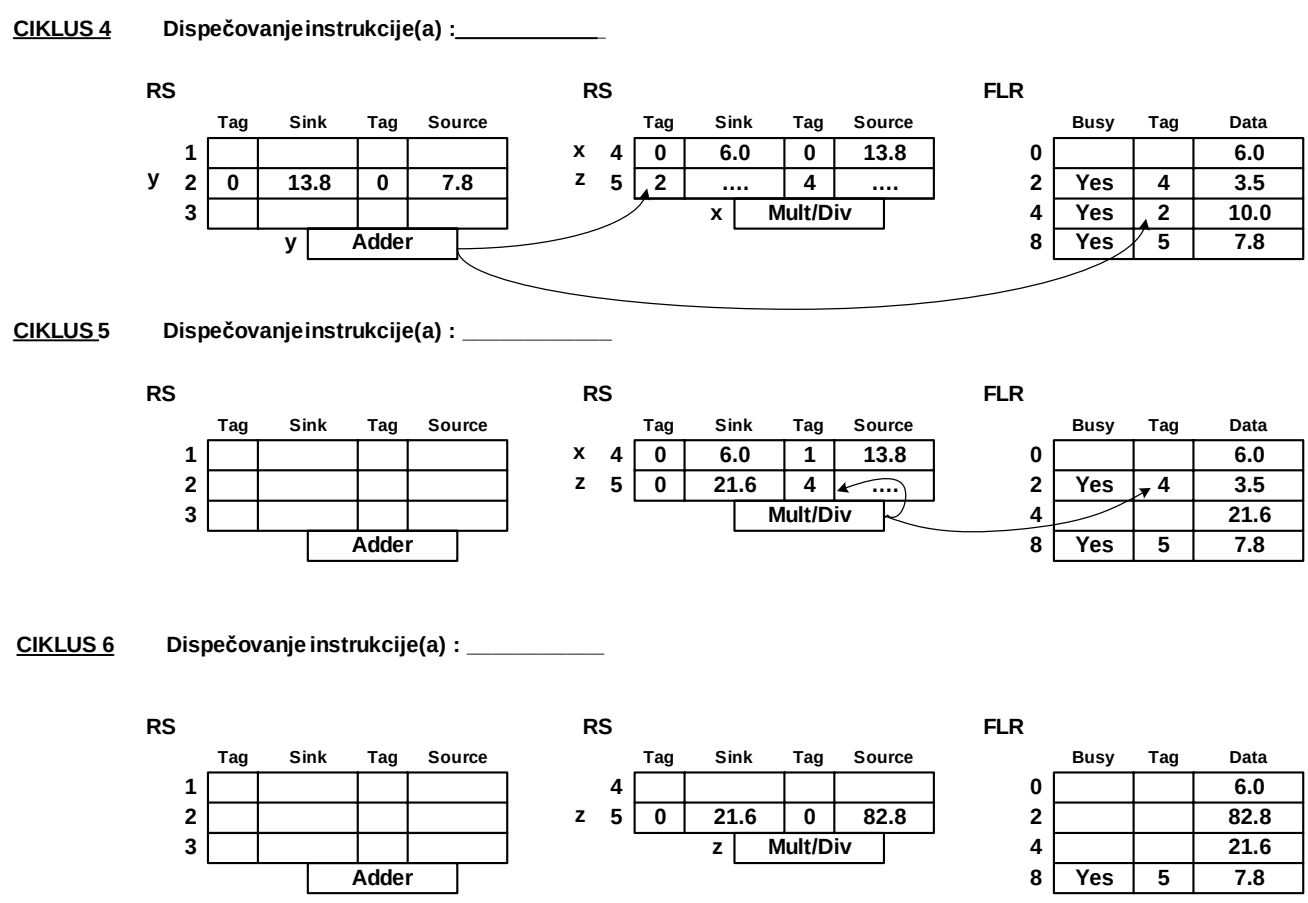


Slika 4.23 Ilustracija *Tomasulo*-ovog algoritma na primeru sekvence instrukcija (prvi deo)

U toku ciklusa #2, prema rezervacionim stanicama 2 i 5 dispečuje se (po redosledu) instrukcije *y* i *z*, respektivno. S obzirom da je instrukciji *y*, kao prvi operand, potreban rezultat koga generiše instrukcija *w*, u trenutku kada se ona dispečuje ka rezervacionoj stanici 2, u *tag* polje prvog operanda se unosi *tag* vrednost 1. Na sličan način, instrukcija *z* koja se dispečuje u rezervacionu stanicu 5, dodeljuju se u oba svoja *tag* polja *tag* vrednosti 2 i 4, što ukazuje da će rezervacione stanice 2 i 4 generisati oba potrebna operanda. S obzirom da je R4 odredište instrukcije *y*, *tag* polje za 4 se ažurira na novu *tag* vrednost 2, što ukazuje da je rezervaciona stanica 2 (tj. instrukcija *y*) sada odgovara za očekivano ažuriranje registra R4. Bit "*busy*" koji se odnosi na R4 ostaje postavljen. Kada se instrukcija *z* dispečuje ka rezervacionoj stanici 5 "*busy*" bit koji se odnosi na R8 se postavlja, a *tag* polje za R8 se postavlja na 5. Na kraju ciklusa #2 *w* završava izvršenje i emituje (postavlja) svoj ID (rezervaciona stanica 1) kao i svoj rezultat na CDB magistrali. Sva *tag* polja koja sadrže *tag* vrednost 1 iniciraće *tag* uparivanje i lečovaće emitovani rezultat prisutan na CDB magistrali. Kod prvog *tag* polja rezervacione stanice (u kome se čuva instrukcia *y*) kao i kod drugog *tag* polja rezervacione stanice 4 (u kome se čuva instrukcija *x*) doći će do uparivanja *tag*-ova. Shodno tome rezultat instrukcije *w* se prosledjuje zavisnim instrukcijama *x* i *y*.

U ciklusu #3, u funkcionalnoj jedinici "*Adder*" počinje izvršenje instrukcije *y*, kao i izvršenje instrukcije *x* u jedinici *Multiply/Divide*. Instrukcija *y* završava izvršenje u ciklusu #4 (vidi sliku 4.24) i emituje svoj rezultat na magistrali CDB zajedno sa *tag* vrednošću 2 (ID svoje rezervacione stanice). Kod prvog *tag* polja u rezervacionoj stanici 5 (u kome se čuva instrukcija *z*) i *tag* polja za R4 dolazi do *tag* uparavanja i smeštanja (prihvatanja) instrukcije *y*. Instrukcija *x* završava svoje izvršenje u ciklusu #5 i emituje svoj rezultat na CDB magistrali zajedno sa *tag* vrednošću 4. Kod drugog *tag* polja u

rezervacionoj stanici 5 (koja čuva instrukciju *z*) kao i *tag* polja za R2 dolazi do uparivanja i prihvatanja rezultata instrukcije *x*. U ciklusu #6, instrukcija *z* počinje svoje izvršenje koje završava u ciklusu #8.

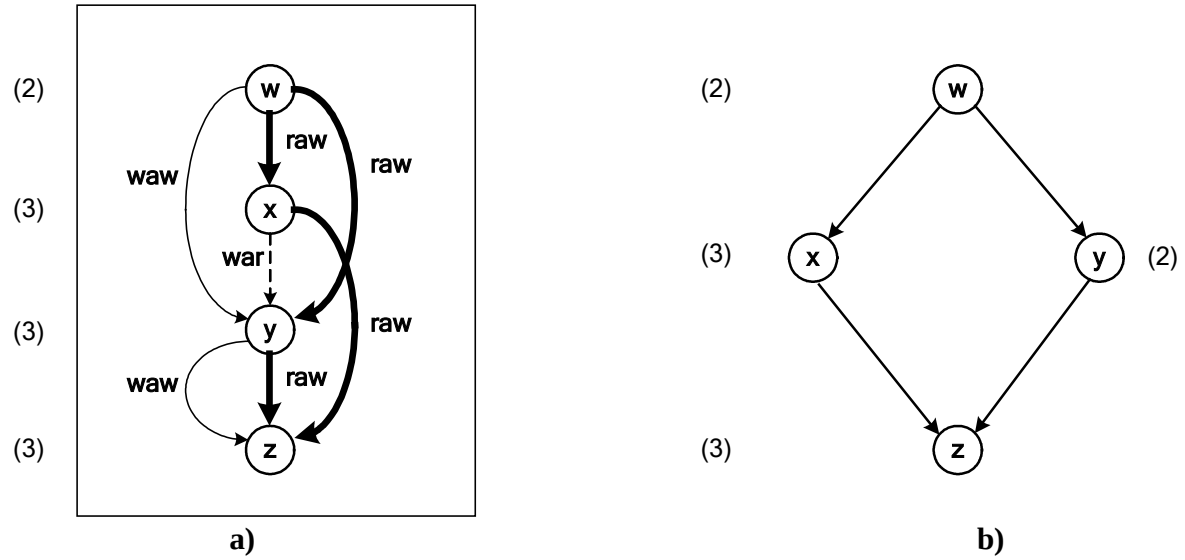


Slika 4.24 Ilustracija *Tomasulo*-ovog algoritma na primeru sekvence instrukcija (drugi deo)

Slika 4.25 prikazuje graf toka upravljanja za gore navedeni primer, a donosi se na sekvencu od četiri instrukcije. Četiri potega izvučena debljom linijom odnose se na četiri prave (*true*) zavisnosti po podacima, dok ostala tri potega se odnose na anti- i izlazne zavisnosti. Instrukcije se dispečuju po programskom redosledu. Anti-zavisnosti se rešavaju kopiranjem operanda u rezervacionu stanicu u trenutku dispečovanja. Zbog toga, nije moguće da prateća instrukcija izvrši upis u registar pre nego što prethodna instrukcija dobija šansu da pročita taj registar. Ako je operand dostupan a nije iskorišćen (*pending*), dispečovana instrukcija će primiti *tag* marker za taj operand. Kada taj operand postane dostupan instrukcija će primiti taj operand putem *tag* uparivanja u svojoj rezervacionoj tabeli.

Nakon što je instrukcija dispečovana, *tag* polje njenog odredišnog registra je upisan sa ID-om rezervacione stanice za tu instrukciju. Kada naredna instrukcija koja koristi isti odredišni registar se dispečuje, isto *tag* polje biće ažurirano sa ID-om rezervacione stanice za ovu novu instrukciju. *Tag* polje registra uvek sadrži ID rezervacione stanice zadnje ažurirane instrukcije. Za slučaj da postoje veći broj instrukcija koje se izvršavaju, a da pri tome one koriste isti odredišni registar, samo će zadnjoj instrukciji biti dozvoiljeno da ažurira registar. Izlazne zavisnosti se implicitno rešavaju čineći ne mogućim da ranije instrukcije ažuriraju registar nakon što su instrukcije, koje kasnije u programu slede, ažurirale isti registar. Ovo dovodi do problema koji se ogleda u tome da nije moguće podržati precizni izuzetak (*exeception*) jer registarsko polje neće korektno evoluirati kroz sva sekvencijalna stanja, tj. registar može potencijalno da izgubi neko medju ažuriranje. Na primer, na slici 4.23 na kraju ciklusa #2 instrukcija *w* treba da ažurira odredišni registar R4. Ipak instrukcija *y* koja koristi isti odredišni registar,

a dispečovana je ranije u toku ciklusa, *tag* polje R4 biće promenjeno sa 1 na 2 što ukazuje na ažuriranje R4 od strane instrukcije *y*. Na kraju ciklusa #2 kada instrukcija *w* emituje svoju *tag* vrednost 1, *tag* polje R4 neće pratiti *tag* uparivanja i neće smestiti svoj rezultat koji pripada instruciji *w*. To znači da će R4 biti ažuriran od strane instruckije *y*. Ipak, ako se javi izuzetak generisan od strane instrukcije *x* precizan izuzetak biće nemoguć iz razloga što registarsko polje nije evoluiralo kroz sva svoja sekvencijalna stanja.



Slika 4.25 DFG-ovi za dati primer sekvence instrukcija: (a) prikazane su sve zavisnosti po podacima; (b) prikazane su prave zavisnosti po podacima

Tomasulo-ov algoritam rešava anti- i izlazne zavisnosti koristeći tehniku preimenojvanja registara. Svaka definicija *FLR* registara inicira preimenovanje tog registra u registar *tag*. Ovaj *tag* se zatim uzima iz ID-a rezervacione stanice koji sadrži instrukciju koja redefiniše taj registar. Na ovaj način efikasno se eliminišu lažne zavisnosti koje mogu dovesti do zastoja u protočnoj obradi. Zbog toga tok upravljanja je striktno odredjen od strane prtave zavisnosti po podacima. Na slici 4.25 b) opisan je DFG koji sadržiu samo prave zavismosti po podacima. Kao što je prikazano na slici 4.25 a) ako se zahteva da se sve četiri instrukcije izvršavaju sekvencijalno sa ciljem da se sačuvaju sve zavisnosti po podacima, uključujući anti- i izlazne zavisnosti, ukupna latencija potrebna za izvršenje ove sekvence instrukcija biće 10 ciklusa, pri zadatim latencijama od dva i tri ciklusa za instruckije sabiranja i množenja, respektivno. Kada se posmatraju samo prave zavisnosti po podacima, analizom slike 4.25 b) zaključujemo da kritični put trajanja 8 ciklusa, tj. put koga čine instrukcije *w*, *x* i *z*. Shodno tome granična vrednost toka upravljanja za ovu sekvencu od četiri instrukcije je 8 ciklusa. Ova granična vrednost se ostvaruje *Tomasulo*-im algoritmom koji je prikazan na slikama 4.23 i 4.24.

4.2.5. Dinamičko *Execution* jezgro

Najveći broj savremenih superskalarnih procesora se realizuje pomoću *out-of-order execute* jezgra koje se nalazi izmedju *in-order front-end*, koji pribavlja i dispečuje instrukcije u programskom redosledu i *in-order back-end* koje kompletira i izvlači (*retire*) instrukcije u programskom redosledu. *Out-of-order execute* jezgro (takodje nazvano dinamičko *execution* jezgro) podseća na princip rada *Tomasulo*-ovog algoritma i može se posmatrati kao mašina koja ima ugradjeni (*embedded*) tok upravljanja, ili *micro-data-flow*, koja pokušava da suzi granice toka upravljanja u toku izvršenja instrukcije. Rad ovakvog dinamičkog *execution* jezgra se može opisati u saglasnosti sa triju faza u

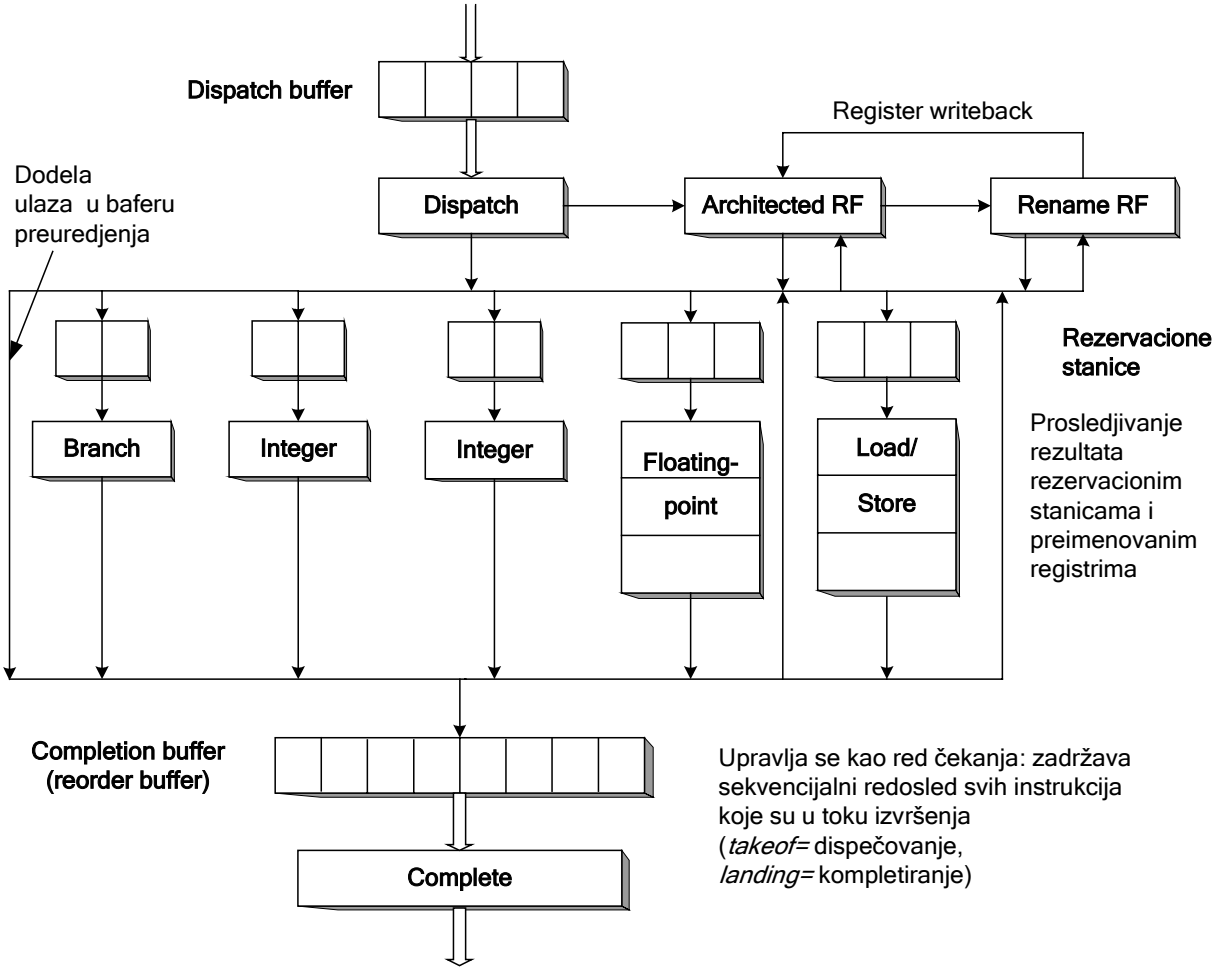
protočnoj obradi, naime to su faze *instruction dispatching*, *instruction execution* i *instruction completion* (vidi sliku 4.26).

Faza dispečovanje instrukcija sastoji se od *preimenovanja* (*renaming*) odredišnih registara, alokaciji (*allocating*) rezervacione stanice i ulaza u baferu preuredjenja, i advansiranju instrukcija od dispeč bafera ka rezervacionim stanicama. Radi lakše prezentacije, u ovom delu usvojićemo da se preimenovanje registara obavlja u stepenu dispeč. Sve redefinicije arhitekturnih registara se preimenuju u preimenovane registre. Zadnja korišćenja ovih redefinicija se dodeljuju ogovarajućim specifikatorima za preimenovanje registara. Na ovaj način se obezbedjuje da svi odnosi tipa proizvoidjač-potrošač budu korektno identifikovani i da sve pogrešne (lažne) registarske zavisnosti budu otklonjene.

Instrukcije u dispeč baferu, na bazi tipa instrucija, se zatim dispečuju u odgovarajuće rezervacine stanice. U ovom slučaju podrazumevamo da se koriste distribucione rezervacione stanice, a da se korišćenje rezervacione stanice odnosi na (više-ulazni) instrukcioni bafer koji je pridružen svakoj funkcionalnoj jedinici, a da se ulaz rezervacione stanice odnosi na jedan od ulaza tog bafera. Simultano sa dodelom ulaza rezervacione stanice u cilju dispečovanja instrukcije obavlja se alokacija ulaza u baferu preuredjenja za te instrukcije. Ulazi bafera preuredjenja se alociraju u saglasnosti sa programskim redosledom.

Obično, da bi se jedna instrukcija dispečovala mora da bude dostupno registar za preimenovanje, ulaz u rezervacionu stanicu, i ulaz u bafer preuredjenja. Ako bilo koji od ova tri resursa nije dostupan dispečovanje instrukcije će se zaustaviti. Aktuelno dispečovanje instrukcija sa ulaza dispeč bafera u ulaze rezervacione stanice se obavlja preko složene mreže za rutiranje. Ako povezljivost mreže za rutiranje je manja od povezljivosti potpune krozbar mreže (što je najčešći slučaj kod realnih dizajna), dolazi do zastoja koji se javljaju zbog sudara (*contetion*) kod korišćenja sprežne mreže.

Faza *Execution* sastoji se od iniciranje izvršenja (*issuing*) spremnih instruckija, izvršenje (*executing*) iniciranih instrukcija, i prosledjivanje (*forwarding*) rezultata. Svaka rezervaciona stanica je odgovorbna za identifikovanje instrukcija koje su psremne za izvršenje i za planiranje (*sheduling*) njihovih izvršenja. Kada se instrukcija prvo dispečuje ka rezervacionoj stanici, svi njeni izvosrni operandi ne moraju biti dostupni pa zbog toga ona mora da čeka u rezervacionoj stanici. Sve instrukcije koje čekaju kontinualno nadgledaju magistrale radi *tag* uparivanja. Kada dodje do *tag* uparivanja što ukazuje na dostupnost operanda koji se očekuje, rezultat koji se emituje (*broadcasted*) se lečuje u ulaz rezervacione stanice. Kada instruckija koja se nalazi u ulazu rezervacione stanice ima raspoložive sve operande, ina postaje spremna za izvršenje i njeno izvršenje se može inicirati u funkcionalnoj jedinici. Ako u datom mašinskom ciklusu u rezervacionoj stanici postoji veći broj spremnih instrukcija, koristi se algoritam za planiranje izvršenja (*scheduling algoritam*) koji u najvećem broju slučajeva inicira prvo izvršenje najstarije instrukcije i usmerava je ka funkcionalnoj jedinici. Ako je samo jedna funkcionalna jedinica povezana na rezervacionoj stanici (kakav je slučaj kod distribuiraniih rezervacioih stanica), tada ta rezervaciona stanica može da inicira izvršenje samo jedne instrukcije po ciklusu.



Slika 4.26 Tok obrade na makro nivou kod dinamičkog izvršenja instrukcija

Nakon što je instrukcija inicirana radi izvršenja u funkcionalnoj jedinici, njeno izvršenje počinje. Latentnost funkcionalnih jedinica može biti promenljiva. Kod nekih FU latentnost je jedan ciklus, a kod drugih postoji fiksna latentnost od većeg broja ciklusa. Odredjenje FU karakteriše latentnost tipa promenljivi broj ciklusa, što zavisi od vrednosti operanada i tipa operacije koja se obavlja. Obično, kod FU čija je latentnost nekoliko ciklusa, nakon što instrukcija počne sa izvršenjem (u protočno organizovanoj funkcionalnoj jedinici), ne postoji u daljem toku izvršenja zaustavljanje te instrukcije u toku protočne obrade jer su sve zavisnosti po podacima već razrešene pre njenog iniciranja izvršenja, a takodje i ne dolazi do sudara zbog korišćenja resursa.

Kada se izvršenje instrukcije završi ona ažurira svoj odredišni *tag* (tj. specifikator preimenovanog registra dodeljen tom odredištu), i aktuelni rezultat se postavlja na magistrali. Sve zavisne instruckije koje čekaju u rezervacionim stanicama iniciraće uparivanje *tag*-a i lečovaće emitovani rezultat koji je prisutan na magistrali. Ovo je način kako instruckija pšrosledjuje svoj rezultat ostalim zavisnim instruckijama bez da se koriste medju koraci koji bi se sastojali u ažuriranju, a zatim čitanju zavisnih registara. Istovremeno sa prosledjivanjem rezultata, RRF koristi *tag* emitovan svima kao indeks i puni emitavani rezultat u selektovani ulaz RRF-a.

Obično ulaz rezervacione stanice se dealocira kada se inicira izvršenje njegove instrukcije kako bi se obezbedilo mesto da se neka od narednih instrukcija dispečuje u njoj. Saturacija rezervacione stanice uzrokuje zastoj u dispečovanju instrukcija. Neke od instrukcija čije izvršenje uzrokuje izuzetak mogu kasnije da zahtevaju ponovno planiranje njihovog izvršenja. Najčešće, kod ovih instrukcija, ulazi rezervacione stanice se ne dealociraju dok ne završi sa izvršenjem, a da pri tome ne izazovu bilo koji izuzetak. Na primer, instrukcija *Load* može da bude uzrok D-keš promašaja za čije opsluživanje je

potreban veći broj ciklusa. Umesto da se *FU* zaustavi, *Load* instrukcija koja je uzrokovala izuzetak se ponovo inicira radi izvršenja iz rezervacione stanice nakon što je promašaj opslužen (*Load* počinje sa izvršenjem iz početka, a ne od trenutka kada je bila prekinuta zbog promašaja). Kod jezgra dinamičkog izvršenja odnos proizvodjač-potrošač je zadovoljen bez potrebe da se čeka na upis, a zatim čitanje zavisnog registra. Zavisni operand se direktno prosledjuje od instrukcije proizvodjač ka instrukciji potrošač sa ciljem da se minimizira latencija koja se javlja zbog prave zavisnosti po podacima. Usvojimo da se instrukcija može inicirati radi izvršenja u istom ciklusu kada ona primi zadnji operand preko *bus*-a za prosledjivanje, i ako ne postoji druga instrukcija koja se takmiči da koristi istu *FU*, tada ova instrukcija treba da je u stanju da počne sa izvršenjem u ciklusu koji neposredno sledi nakon dostupnosti svih njenih operanada. Stoga, ako postoje adekvatni resursi tako da se zastoji zbog strukturne zavisnosti ne jave, tada kôd dinamičkog izvršenja treba da bude u stanju da se približii granici koja je diktirana tokom podataka.

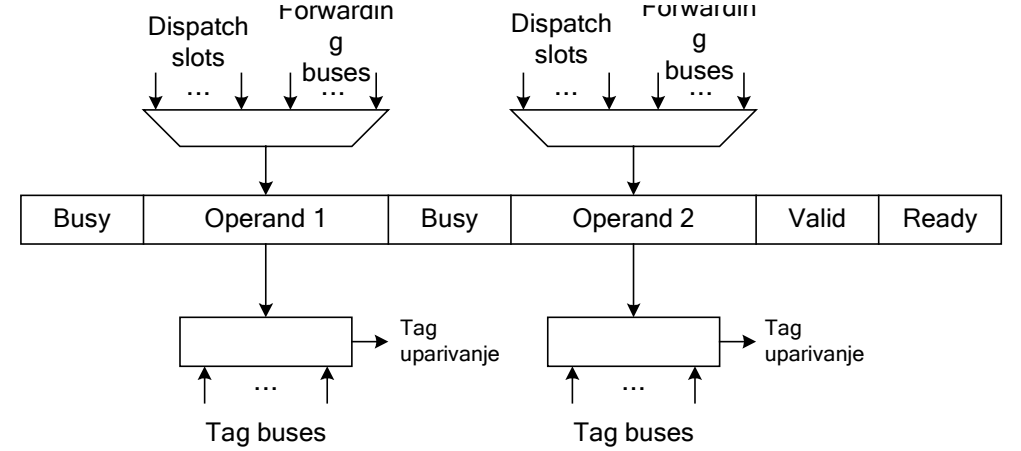
4.2.6. Rezervacione stanice i bafer preuredjenja

Osim *FU*-a, kritične komponenete kod jezgra dinamičkog izvršenja su *rezervaciona*(*e*) *stanica*(*e*) i *bafer preuredjenja*. Rad ovih bafera diktira funkcionisanje jezgra dinamičkog izvršenja. Ukazaćemo sada na stavke koje su važne za implementaciju rezervacine stanice i bafera preuredjenja. Analiziraćemo njihovu organizaciju i ponašanje, a posebno ćemo se osvrnuti na punjenje i pražnjenje ulaza rezervacione stanice i bafera preuredjenja.

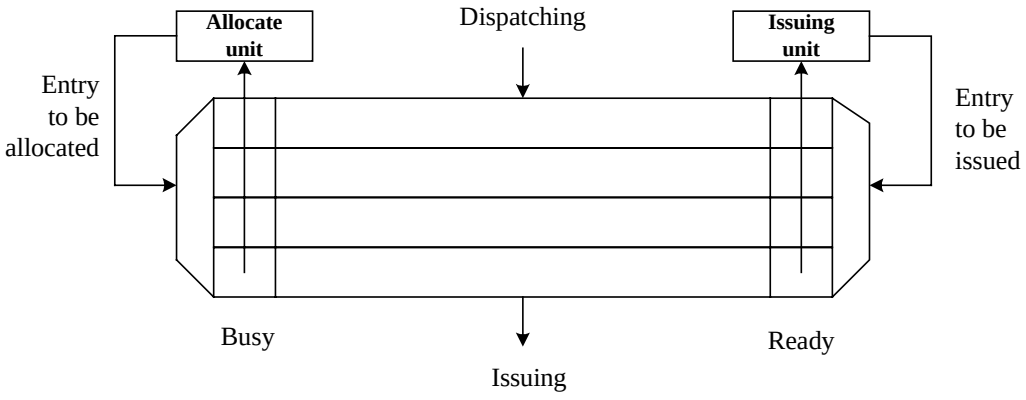
Postoje tri zadatka koja su tipična za rad rezervacione stanice, a to su: dispečovanje (*dispatching*), čekanje (*waiting*), i iniciranje izvršenja (*issuing*). Jedna tipična rezervaciona stanica prikazana je na slici 4.27 b), a različita polja u okviru ulaza rezervacione stanice su prikazana na slici 4.27 a). Svaki ulaz ima *busy* bit, koji ukazuje da je taj ulaz alociran, i *ready* bit, koji ukazuje da instrukcija na tom ulazu raspolaže sa svim operandima. Dispečovanje podrazumeva punjenje instrukcije iz dispeč bafera u ulaz rezervacione stanice. Obično dispečovanje instrukcije čine sledeća tri koraka:

a) izabiranje slobodnog, tj. *not-busy*, ulaza rezervacione stanice,
b) punjenje operanda i/ili *tag*-ova u selektovani ulaz, i
c) postavljanje *busy* bita tog ulaza.

Selekcija slobodnog ulaza se zasniva na *busy* bitovima i obavlja se od strane jedinice za alokaciju (dodelu). Alokaciona jedinica ispituje sve *busy* bitove i selektuje jedan od *non-busy* ulaza kako bi se polju koje prati *non-busy* ulaz dodelio operand. Ovo se implementira pomoću kodera prioriteta. Nakon što je ulaz alociran operandi/*tag*-ovi instrukcije se pune u ulaz. Svaki ulaz ima dva polja za operande, pri čemu je svakom pridruženo polje *valid* bit. Ako polje operanda sadrži aktuelni operand, tada je *valid* bit postavljen. Ako polje sadrži *tag* što ukazuje da se na operand čeka (*pending operand*), tada njegov *valid* bit je resetovan i mora da se sačeka da operand bude prosledjen (dobavljen). Nakon što je ulaz alociran, njegov *busy* bit se postavlja.



a)



b)

Slika 4.27 Mehanizmi rezervacione stanice: a) ulaz rezervacione stanice; b) dispečovanje i iniciranje izvršenja iz rezervacione stanice.

Instrukcija koja čeka na dostupnost operanda mora da čeka u rezervacionoj stanici. Kada ulaz rezervacione stanice čeka na operand (*pending operand*) on mora kontinualno da nadgleda *tag* magistralu(e). Kada dodje do uparivanja *tag*-ova polje operanda lečuje prosledjeni rezultat i setuje bit *valid*. Kada su *valid* bitovi za oba operanda setovana, setuje se *ready* bit, što ukazuje da instrukcija ima raspoložive sve izvorne operande i može da počne njeno iniciranje izvršenja. Ovo se obično naziva "budjenje " instrukcije (*instruction wake-up*).

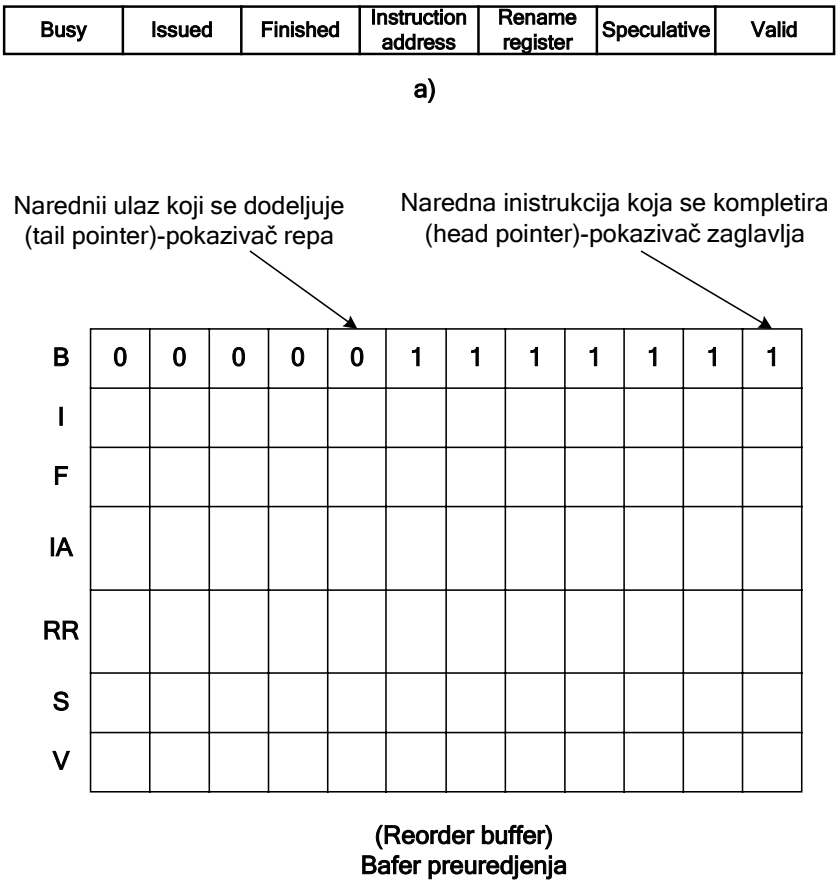
Korak iniciranja izvršenja instrukcije (*issuing step*) je odgovoran za selekciju spremne instrukcije u rezervacionoj stanici i njenom iniciranju izvršenja u *FU*-i. Ovo se obično naziva selektovanje instrukcije (*instruction select*). Sve spremne instrukcije se identifikuju na osnovu toga da li su *ready* bitovi postavljeni ili ne. Selektovanje spremne instrukcije se obavlja od strane jedinice za iniciranje izvršenja (*issuing unit*) na osnovu heurističkog pristupa u planiranju izvršenja instrukcija (vidi sliku 4.27b)). Heuristika može biti zasnovan na programskom redosledu ili na tome koliko dugo je spremna instrukcija je čekala u rezervacionoj stanici. Obično kada se u *FU*-i inicira izvršenje instrukcije, njen ulaz rezervacine tabele se dealocira, a odgovarajući *busy* bit se resetuje.

Rezervacione tabele velikog obima mogu biti veoma kompleksne za implementciju. Na svojoj ulaznoj strani, rezervaciona stanica mora da podržava što je moguće veći broj izvorišta, uključujući sve

dispeč slotove i magistrale prosledjivanja (vidi sliku 4.27 a)). Mreža za rutiranje podataka na ulaznoj strani može biti veoma kompleksna. U toku koraka čekanja (*waiting*) sva polja operanda rezervacione stanice koja treba da prihvate operande koji se očekuju (*pending operand*) mora kontinualno da kompariraju odgovarajuće *tag*-ove u odnosu na informaciju koja je prisutna na većem broju *tag* magistrala. Ovo se ostvaruje pomoću asocijativnog pretraživanja kroz sve ulaze rezervacione stanice uključujući tu veći broj ključeva (*tag* magistrala). Ako je broj ulaza mali tada implementacija i nije složena. Ali, ako se broj ulaza povećava, značajno se povećava kompleksnost. Ovaj deo hardvera se standardno naziva logika za budjenje (*wake up logic*). Ako se broj ulaza poveća, tada se komplikuje kako jedinica za inciranje izvršenja instrukcija tako i logika za heurističko planiranje izvršenja instrukcija koja treba da selektuje najspremniju instrukciju čije izvršenje treba da započne. Ovaj deo hardvera se standardno naziva logika za selekciju (*select logic*). U nekom proizvoljnom mašinskom ciklusu može da postoji veći broj spremnih instrukcija. Selekciona logika mora da odredi "najbolju" instrukciju radi iniciranja izvršenja. Kod *SSPM*-e rezervaciona stanica može potencijalno da podrži po jednom mašinskom ciklusu iniciranje izvršenja većeg broja instrukcija, pri čemu logika za selekciju mora da odabere najbolji podskup instrukcija od svih mogućih spremnih instrukcija.

Bafer preuredjenja sadrži sve instrukcije koje su "*u letu*" (*in flight*), tj. sve instrukcije koje su dispečovane ali još nisu arhitekturno kompletirane. To su ustvari sve instrukcije koje čekaju u rezervaionim stanicama, izvršavaju se u funkcionalnim jedinicama, kao i one koje su završile sa izvršenjem ali čekaju radi kompletiranja u programskom redosledu. Status svake instrukcije u baferu preuredjena se može pratiti pomoću nekoliko bitova u svakom od ulaza u baferu preuredjenja. Svaka instrukcija može da bude u jedno od sledećih stanja, tj. da čeka na izvršenje, da se izvršava, i da je završila izvršenje. Statusni bitovi se ažuriraju kako instrukcija prolazi od jednog stanja u narednom. Dodatni bit može se takodje koristiti da ukaže da li je instrukcija spekulativna (na prediktovanom putu) ili nije. Ako spekulacija može da prodje kroz nekoliko grananja, koriste se dodatni bitovi koji mogu da posluže da identifikuju kojem spekulativnom osnovnom bloku instrukcija pripadaju. Kada je grananje razrešeno, spekulativne instrukcije mogu postati nespekulativne (ako je predikcija korektna) ili invalidne (ako predikcija nije korektna). Samo završene i nespekulativne instrukcije mogu da se kompletiraju. Instrukcija koja je bila invalidna se ne kompletira kada napusti bafer preuredjenja.Slika 4.28 prikazuje polja koja se tipično sreću kod ulaza u bafer preuredjenja; na ovoj slici polje preimenovanog registra je takodje prikazano.

Bafer preuredjenja se upravlja kao kružni red čekanja koristeći pokazivač zaglavlje (*head*) i pokazivač rep (*tail*) vidi sliku 4.28 b). pokazivač rep se advansira kada se ulazi bafera preuredjenja alociraju u toku dispečovanja instrukcija. Broj ulaza koji se u jednom ciklusu može alocirati odredejn je propusnošću dispečovanja. Instrukcije se kompletiraju sa zaglavlja reda čekanja. Sa zaglavlja reda čekanja sve one instrukcije koje su završile sa izvršemnjem mogu da se kompletiraju onoliko brzo koliko propusnost *completion* dozvoljava. Propusnost *completion* je odredejna kapacitetom druge sprežne mreže i portova koji su dostupni za *writeback* u registar. Jedna od kritičnih stavki predstavlja broj portova za upis u arhitekturno *RF* polje koji su potrebni da podrže prenos podataka od preimenovanih registara (ili ulaza bafera preuredjenja ako se oni koriste kao registri za preimenovanje) ka arhitekturnim registrima. Kada se instrukcija kompletira, njen preimenovani registar i njen ulaz u bafer preuredjenja se dealociraju.Pokazivač zaglavlja bafera preuredejenja se takodje ažurira na adekvatan način. To znači da se bafer preuredjenja može posmatrati ako srce ili centralni blok upravljanja jezgra dinamičkog izvršenja jer status svih instrukcija "u letu" (*in-flight,* tj. koje se izvršavaju) se može pratiti od strane bafera preuredjenja.



Slika 4.28 a) ulaz bafera preuredjenja; b) organizacija bafera preuredjenja

Moguće je kombinovati rezervacionu(e) stanicu(e) i bafer preuredjenja u jedinstvenu strukturu koju nazivamo instrukcioni prozor (*instruction window*), koji upravlja svim instrukcijama "u letu". Pošto u trenutku dispečovanja, za svaku instrkciju ulaz u rezervacionu stanicu kao i ulaz u bafer preuredjenja, se moraju alocirati, one se mogu kombinovati kao jedan ulaz u instrukcioni prozor. Stoga instrukcije se prvo dispečuju u instrukcioni prozor, nakon toga ulazi instrukcionog prozora nadgledaju *tag* magistrale za operande koji se očekuju (*pending operands*), zatim rezultati se prosledjuju u instrukcioni prozor, posle toga instrukcije se iniciraju radi izvršenja iz instrukcionog prozora u trenutku kada su spremne, i na kraju instrukcije se kompletiraju iz instrukcionog prozora. Obim instrukcionog prozora odredjuje maksimalan broj instrukcija koje se mogu simultano "u letu" naći u samoj mašini, a shodno tome time je odredjen stepen paralelizma na nivou instrukcija koji se može ostvariti od strane te mašine.

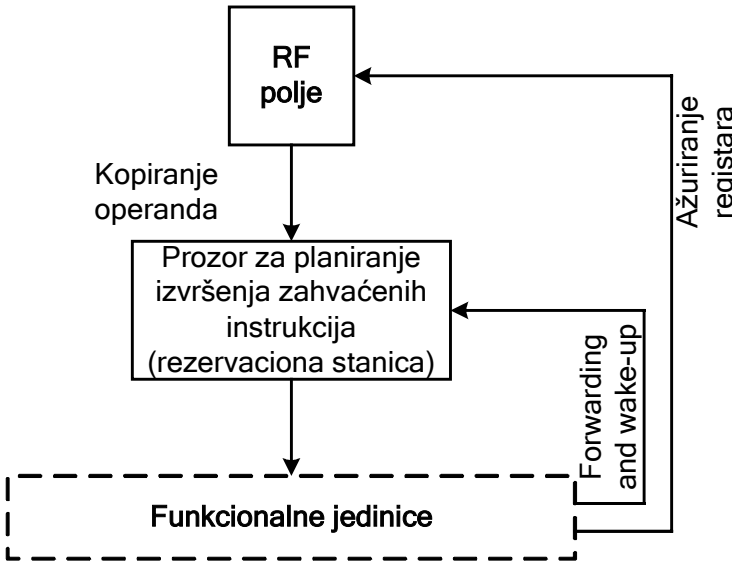
4.2.7. Dinamički planer instrukcija

Dimanički planer instrukcija (*Dynamic Instruction Scheduler-DIS*) predstavlja srce jezgra dinamičkog izvršenja. Termin *DIS* uključuje prozor instrukcija kao i odgovarajuća *wake-up* i selekciona logika. Tekuće, postoje dva dizajn stila koja se odnose na realizaciju *DIS*-a. Prvi je sa-zahvatanjem-podataka (*with data capture-WDC*), a drugi je bez-zahvatanja-podataka (*without data capture-WoDC*).

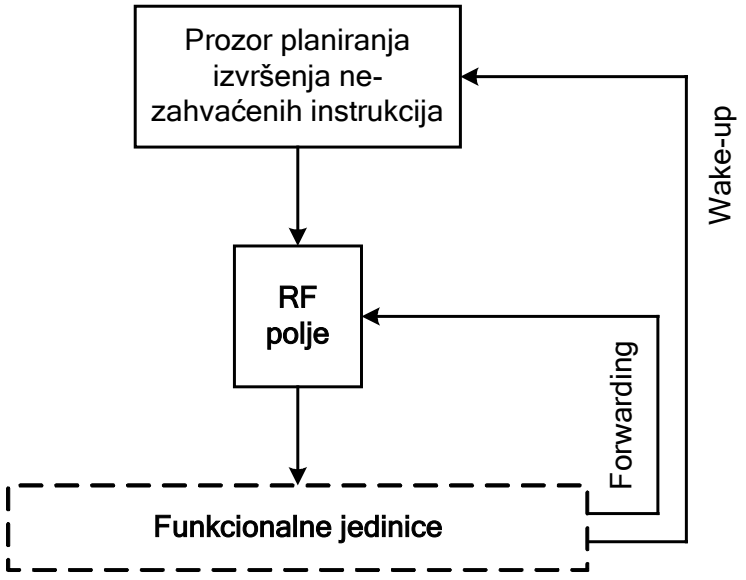
Na slici 4.29 prikazan je planer tipa *WDC*. Kod ovog tipa planera, u trenutku dispečovanja instrukcije, oni operandi koji su psremni se kopiraju iz *RF* polja (bilo arhitekturnog ili fizičkog) u instrukcioni prozor, pa se zbog toga koristi termin *zahvaćeni-podaci*.Za operande koji nisu spremni, *tag*-ovi se kopiraju u instruckoni prozor i koriste se za lečovanje operanada kada se oni prosledjuju od strane FU-ova. Rezultati se prosledjuju instrukcijama koje čekaju na te operande u instrukcionom prozoru. U suštini, rezultat koji se prosledjuje instrukciji koja treba da se aktivira treba da bude u saglasnosti sa *Tomasulo*-ivim algoritmom. Poseban put za prosledjivanje je potreban za ažuriranje *RF* polja tako da naredne zavisne instrukcije mogu da dograbe svoje izvorne operande iz *RF* poja kada se one dispečuju.

Kod novih mikroprocesora koristi se drugačiji dizajn stil koji se ne bazira na *zahvatanju-podataka* u stepenu za *planiranje* (vidi sliku 4.29 b)). Kod ovog stila, čitanje registra se obavlja nakon *planiranja*, tj. nakon iniciranja izvršenja instrukcije u *FU*-i. Dispečovanje instrukcije ne prati kopiranje operanada u instrukcioni prozor; tj. samo se *tag*-ovi (ili pokazivači) operanada pune u prozor. Planer i dalje obavlja uparivanje *tag*-ova kako bi *wake-up* (probudio) spremen instrukcije. Ipak se samo rezultati sa *FU* –a prosledjuju *RF* polju. Sve spremne instrukcije koje su inicirane radi izvršenja dobijaju svoje operande direktno iz *RF* polja pre nego što počne njihovo izvršenje. U suštini, aktivnostri tipa prosledjivanje rezultata kao i "budjenje" instrukcije su medjusobno razdvojene. Da bi se instrukcija "probudila" neophodno je da se *tag* prosledi planeru. Kod *WoDC* planera, obim instrukcionog prozora se može značajno redukovati, pa zbog toga znatno obimniji put za prosledjivanje rezultata planeru nije potreban.

Postoji čvrsta sprega izmedju preimenovanja registara i planiranja izvršenja instrukcija. Kao što smo napomenuli tranije, jedna od svrha da se obavi dinamičko preimenovanje registara sastoji se u eliminisanju lažnih zavisnosti koji se javaljaju zbog višestrukog korišćenja registara. Druga svrha je da se uspostavi odnos proizvodjač-potrošač izmedju dve zavisne instrukcije. Prava zavisnost po podacima se odredjuje pomoću zajedničkog specifikatora za preimenovanje registara koji je definisan instrukcijama proizvodjač-potrošač. Specifikator za preimenovanje registara može da funkcioniše kao *tag* za prosledjivanje rezultata. Kod planera koji *ne-koristi-zahvatanje-podataka* (*non-data-captured-scheduler*) (vidi sliku 4.29 b)), radi izbavljanja izvornih operanada iz *RF* polja koriste se registarski specifikatori; (odredišni) registarski specifikatori se koriste kao *tag*-ovi za "probudjivanje" zavisnih instrukcija u planeru. Za tip planera koji koristi zahvatanje-podataka (*data-captured-scheduler*) (vidi sliku 4.29 a)), *tag*-ovi koji se koriste za prosledjivanje i "probudjivanje" instrukcije ne treba da predstavljaju aktuelni registarski specifikator. *Tag*-ovi se uglavnom koriste da identificiraju odnose proizvodjač-potrošač koji postooji izmedju dve zavisne instrukcije i kojima se može proizvoljno pristupati. *Tomasulo*-ov algoritam koristi ID-ove rezervacione stanice kao *tag*-ove za prosledjivanje rezultata zavisnim instrukcijama kao i za ažuriranje arhitekturnih registara. Ne postoji eksplicitno preimenovanje registara koje uključuje preimenovanje fizičkih registara.



a)



b)

Slika 4.29 Dinamičko planiranje izvršenja instrukcija: (a) sa zahvaćenim podacima; (b) sa nezahvaćenim podacima

4.2.8. Druge tehnike za regulisanje toka podataka prema registrima

Za dugi vremenski period ograničenje koje postoji zbog toka podataka smatralo se da predstavlja apsolutno teoretsko ograničenje, a dostizanje vrednosti te granice smatra se vršni cilj u postizanju prerformansi. Intenzivna istraživanja na polju *data-flow* arhitektura kao i *data-flow* mašina vršena su zadnje tri decenije. Ograničenja tipa *data-flow* (toka-podataka) pretpostavljaju da su prave zavisnosti apsolutne i da se ne mogu prevazići. Ono što je interesantno, kasnih 60-tih i početkom 70-tih

godina slične pretpostavke su bile usvojene da važe i za upravljačke zavisnosti. U opštem slučaju se stajalo na stanovištu da su upravljačke zavisnosti apsolutne i da kada su u pitanju instrukcije uslovnog grananja ne postoji drugi izbor nego mora da se čeka da se uslovno grananje izvrši pre nego što se produži s narednom instrukcijom zbog neodredjenosti od stvarnog toka upravljanja. Nakon ovog perioda, učinjeni su ogromni napori koji su pre svega orijentisani ka tehnikama za predikciju grananja. Uslovna grananja kao i prateće upravljačke zavisnosti ne predstavljaju više apsolutne barijere i mogu se veoma često premostiti spekulisanjem smera ciljne darese instrukcije grananja. Osnova koja čini ovu spekulaciju mogućom je da veoma često ishod instrukcije grananja bude zaista predvidiv. Nakon 1995. godine istraživači su često puta postavljali pitanja o apsolutnosti prave zavisnosti po podacima.

Nekoliko istraživača 1996. godine predložilo je koncept predikcije vrednosti (*value prediction*). Prvi od tih članaka, autora *Lipasti* i dr., fokusirao se na predikciji *load* vrednosti bazirajući se na opservaciji da najčešće vrednosti koje se pune od strane pojedinih statičkih *Load* instrukcija mogu biti veoma predvidljive. Jedan drugi članak generalizuje ovu istu osnovnu ideju kod predikcije rezultat *ALU*-ovih instrukcija. Eksperimentalni podaci zasnovani na realnim ulaznim skupovima podataka ukazuju da rezultati generisani od strane velikog broja instrukcija mogu zaista biti predviljivi. Pojam lokalne-vrednost (*value locality*) ukazuje da odredjene instrukcije teže repetitivno da generišu isti (mali) skup (ponekad jedinstven) rezultantnih vrednosti. Praćenjem rezultata generisanih od strane ovog skupa instrukcija, na osnovu istorije prethodno generisanih vrednosti, moguće je prediktovati generisanje budućih vrednosti. Na osnovu ovih radova kasnije je preložen veći broj rešenja i projektovan veći broj prediktora vrednosti. Novija istraživanja, koja se odnose na hibridne prediktore vrednosti pokazuju da se može sotvariti tačnost predikcije od 80%, realistička rešenja mogu da ostvare *IPC* poboljšanja u opsegu od 8.6 %– 23 % za SPEC benčmarkove.

Kada je rezultat instrukcije korektno prediktovan koristeći prediktovanu-vrednost, što se obično obavlja od strane stepena *Fetch*, naredna zavisna instrukcija može da počne izvršenje koristeći ovu spekulativnu vrednost bez da čeka na aktuelno dekodiranje i izvršenje vodeće instrukcije. Na ovaj način efikasno se eliminiše serijsko ograničenje koje postoji zbog prave zavisnosti po podacima ozmedju dve instrukcije. Na ovaj način ograničenja zbog zavisnosti u *DFG*-u se efikasno eliminišu ako je obavljenja korektna predikcija vrednosti. To znači da predikcija vrdnosti predstavlja jedan potencijal koji premašuje granicu klasičnog toka podataka. Naravno da je validacija i dalje neophodna kako bi se osigurali da je predikcija korektna, a sad to predstavlja novo ograničenje za postizanje veće propusnosti kod izvršenja instrukcija. Predikcija vrednosti je zaosta efikasna i dovodi do povećanja performansi mašine ako je pogrešna predikcija retka, a cena koja treba da se plati zbog pogrešne predikcije mala (tj. nula ili jedan ciklus) i ako latencija validacije je manja od prosečne latencije izvršenja uinstrukcija. To znači da efikasna implementacija predikcije vrednosti je ključno za obezbedjenje efikasnog poboljšanja performansi.

Jedna druga novo predložena ideja se bazira na dinamičkom ponovnom korišćenju instrukcija (d*ynamic instruction reuse*). Slično konceptu lokalna-vrednost (*value locality*), opservacije realnih programa ukazuju da se veoma često iste sekvence instrukcija repetitivno izvršavaju koristeći pri tome isti skup ulaznih podataka. Ovo rezultira redudantnom izračunavanju koje se obavlja od strane mašine. Tehnike koje višestruko koriste dinamičko izvršenje instrukcija pokušavaju da prate trag ovih redudantnih izračunavanja, detektuju ih, a prethodne rezultate koriste bez da obave ova redudantna izračunavanja. Ove tehnike su nespekulativne pa zbog toga kod njih nije potrebna validacija. Dok se predikcija vrednosti (*value prediction*) može posmatrati kao postupak za validaciju odredjenih potega zavisnosti u *DFG*-u, tehnike za dinamičko ponovno korišćenje instrucija pokušavaju da otklone kako čvorove tako i potege podgrafa iz *DFG*-a. Najnovija istraživanja pokazuju da se eliminacijom redudantnih izračunavanja mogu ostvariti značajna poboljšanja kada su programi kreirani na nekom od funkcionalnih jezika. Takodje novija istraživanja ukazuju na slične tendencije koje su prisutne kod realnih programa a odnose se na redudantna izračunavanja. No ipak treba naglasiti da se u skorije vreme očekuju još značajniji rezultati na ovom polju.

4.3. Tehnike za ubrzanje toka podataka prema memoriji

Instrukcije koje se obraćaju memoriji odgovorne su za kopiranje podataka izmedju glavne memorije i *RF* polja, ali su one esencijalne i sa aspekta podrške rada izvršenja *ALU* instrukcija. Registarski operandi koji su potrebni *ALU* instrukcijama moraju se prvo napuniti iz memorije. Sa ograničenim brojem registara nije moguće da se u toku izvršenja programa, svi operandi čuvaju u *RF* polju. Kompajler generiše dodatno kôd (*spill code*) kako bi privremeno smestio odredjene operande u glavnu memoriju i ponovo ih pribavio kada su oni potrebni. Ovaj dodatni kôd se implementira pomoću instrukcije *Load* i *Store*. Obično kompilator samo skalarne promenljive dodeljuje dodeljuje u registre. Složene strukturne podatke, kakva su polja i povezane liste, koje premašuju obim *RF* polja, obično se čuvaju u glavnoj memoriji. Da bi se obavile operacije nad ovakvim strukcurama podataka neophodne su instrukcije tipa *Load* i *Store*. Efikasno procesiranje instrukcija *Load* i *Store* može značajno da minimizira potrebno vreme (*overhead*) koje je potrebno da bi se kopirali podaci izmedju glavne memorije i registarskog polja.

Procesiranje instruckija *Load/Store* kao i rezultujući protok podataka prema memoriji može postati usko grlo ukupnih performansi mašine imajući pre svega u vidu dugu latentnost kod izvršenja memorijskih instrukcija. Duga latentnost kod izvršenja *Load/Store* instrukcija rezulruje pre svega zbog potrebe da se izračunava memorijska adresa i potrebe da se pristupi memorijskoj lokaciji. Da bi se podržao rad sa virtuelnom memorijom. izračunatu memorijsku adresu (nazvanu virtuelnu adresu) neophodno je, pre nego što se može pristupiti fizičkoj memoriji, prevesti u fizičku adresu. Keš memorije su veoma efikasne kada je u pitanju redukcija efektivne latentnosti kod pristupa glavnoj memoriji. Šta više, razvijene su različite tehnike za redukciju ukupne latentnosti i povećanje propusnosti kod procesiranja instrukcija tipa *Load/Store*.

4.3.1. Instrukcije za pristup memoriji

Izvršenje instrukcija za prenos podataka ka/iz memorije se odvija i tri koraka: generisanje memorijske adrese, prevodjenje memorijske adrese, i pristup podataka u memoriju. Mi ćemo prvo analizirati ova tri koraka, a zatim opisati procesiranje *Load/Store* instrukcija kod *SSPM*-a.

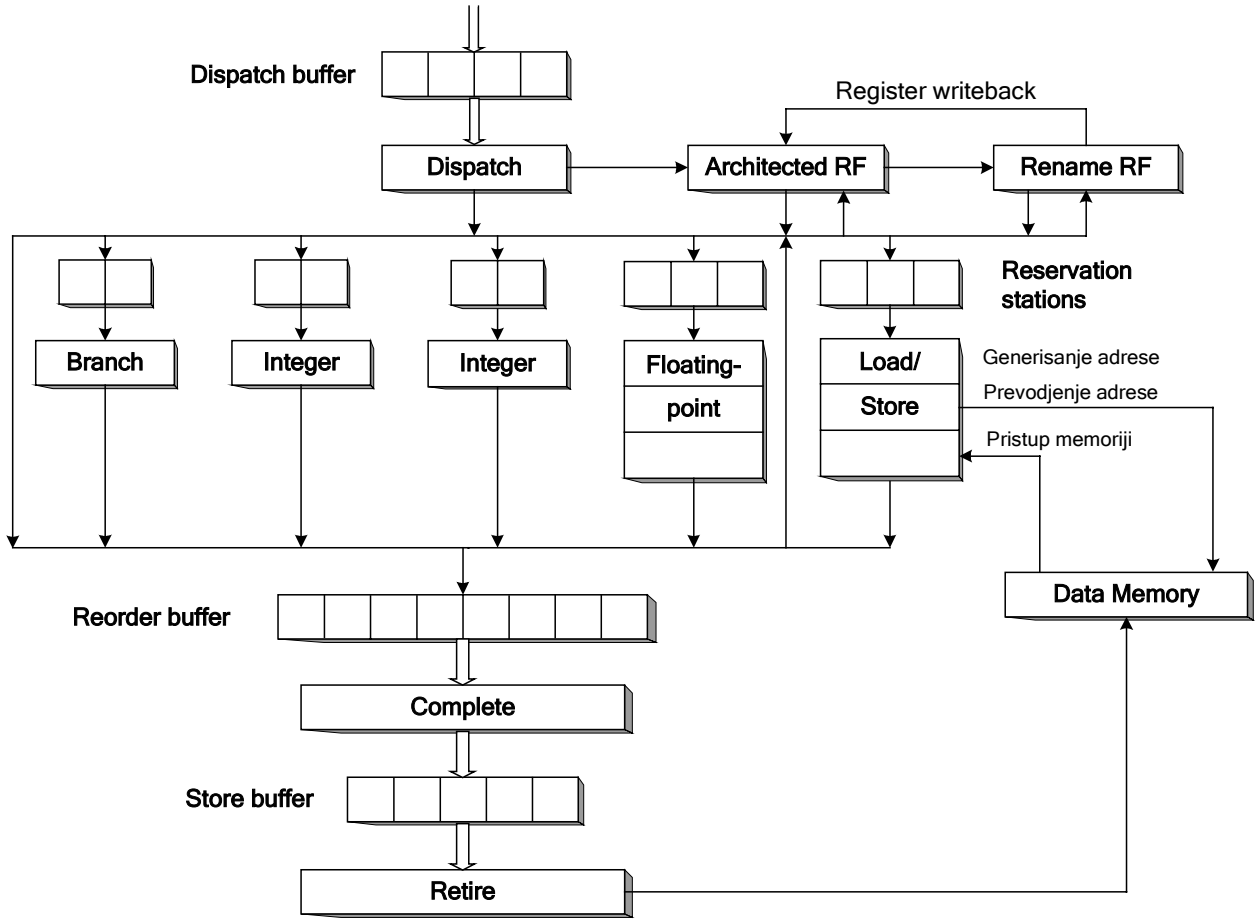
Od strane ISA, *RF* polje i glavna memorija su odredjene za čuvanje podataka. *ISA* definiše memoriju kao skup od 2^n memorijskih lokacija kojima se može proizvoljno pristupati, tj. svaka memorijska lokacija se identifikuje pomoću *n*-to bitne adrese i može joj se direktno pristupiti istom latentnošću. Upravo kao i arhitekturno *RF* polje, i glavna memorija predstavlja jednu arhitekturnu celinu koja je vidljiva softverskim instrukcijama. Ipak nasuprot *RF* polju, adresa koja identifikuje pojedinu memorijsku lokaciju, ne predstavlja eksplicitni deo koji se pamti od strane formata instrukcije. Umesto toga, memorijska adresa se obično generiše na sonovu sadržaja specificiranog registra i ofseta. To znači da je za generisanje adrese potrebno obaviti pristup specificiranom registru, a zatim nad tim sadržajem dodati ofset vrednosti.

Pored generisanja adrese, za slučaj kada je u sistemu implementirana virtuelna memorija, neophodno je obaviti i prevodjenje (*translation*) adrese. Realno instalirana glavna memorija predstavlja deo virtuelnog adresnog prostora sistema i vidi se od strane svakog programa kao privatni adresni prostor za taj program. Fizička memorija koja je implementirana u mašinu odgovara fizičkom adresnom prostoru. Fizički adresni prostor je manji u odnosu na virtuelni i može da bude deljiv za veći broj programa. Virtuelna memorija je mehanizam koji preslikava virtuelni adresni prostor programa u fizički adresni prostor mašine. Ovakvim adresnim preslikavanjem, virtuelna memorija je u stanju da podrži izvršenje programa sa virtuelnim adresnim prostorom koji je veći u odnosu na fizički adresni prostor, kao i multiprogramsku paradigmu putem preslikavanja većeg broja virtuelno adresnih prostora u isti fizički adresni prostor. Mehanizam preslikavanja uključuje prevodjenje izračunate efektivne

adrese, tj. virtuelne adrese u fizičku adresu koja se može koristiti za pristup fizičkoj memoriji. Ovaj mehanizam obično se implementira koristeći tabelu preslikavanja, a proces prevodjenja (transliranje adresa) se vrši pomoću *lookup* tabela (tabela pretraživanja).

Treći korak kod procesiranja *Load/Store* instrukcija se odnosi na pristup memoriji. Kod instrukcije *Load* podaci se čitaju iz memorijske lokacije i smeštaju u registar, dok se instrukcijom *Store* sadržaj registra smešta u memorijsku lokaciju. Dok se prva dva koraka koja se odnose na generisanje i prevodjenje adrese, (kod *Load* i *Store* instrukcije), obavljaju na identičan način, treći korak, kod *SSPM*-a, se izvršava na različit način za *Load* i *Store*.

Na slici 4.30 prikazan je redosled pojavljivanja ova tri koraka u tri protočna stepena. Za obe, *Load* i *Store*, instrukcije usvajamo da se koristi registarsko adresni način rada sa ofsetom. Kod *Load* instrukcije. onog trenutka kada je operand adresnog registra dostupan, isti se predaje protočnoj funkcionalnoj jedinici, pa se efektivna adresa generiše od strane prvog protočnog stepena, *Store* instrukcija pre nego što se inicira radi izvršenja mora da sačeka na dostupnost oba operanda, prvo onaj koji se čuva u adresnom registru , a zatim na podatak koji se pamti u nekom od registra.



Slika 4.30 Procesiranje instrukcija *Load/Store*

Nakon što je prvi protočni stepen generisao efektivnu adresu, drugi protočni stepen prevodi ovu virtuelnu adresu u fizičku adresu. Obično se ovo izvodi pristuanjem TLB-u (*transtation lookahead buffer*) koji je u suštini hardversko-kontrolisana tabela, a ostvaruje preslikavanje virtuelnih u fizičke adrese. *TLB* je u suštini keš tabela straničenja (*page table*) koja se nalazi u glavnoj memoriji. (U sekciji 4.3.2 dato je više detalja koji se odnose na tabelu straničenja i *TLB*). U toku rada može da se desi da prevedena virtuelna adresa pripadne nekoj stranici za koju u *TLB*-u ne postoji informacija o

preslikavanju. Ovu situaciju nazivamo *TLB* promašaj. Ako pojedino preslikavanje egzistira u tabeli straničenja, tada se informacija o preslikavanju može izbaviti pristupanjem tabeli straničenja u glavnoj mamoriji. Nakon što se informacija o preslikavanju kod promašaja izbavi i napuni u *TLB*, prevodjenje se završava. Moguće je da se desi i sledeća situacija: Da informacija o preslikavanju ne bude rezidentna čak i u tabeli straničenja, što znači da pojedina stranica kojoj se obraćamo nije preslikana i nije rezidentna u glavnoj memoriji. Ovo će uzrokovati grešku straničenja (*page fault*), a da bi se dobavila stranica koja nedostaje, zahteva se pristup disku. Kao rezultat inicira se programski izuzetak (*program exception*), što dovodi do suspenzije izvršenja tekućeg programa.

Nakon uspešnog prevodjenja adrese u drugom protočnom stepenu, u toku trećeg protočnog stepena, instrukcija *Load* pristupa podatku koji se čuva u memoriji. Na kraju ovog mašinskog ciklusa, podaci se izbavljaju iz memorije za podatke i upisuju u preimenovani registar ili u bafer preuredjenja. U ovom trenutku instrukcija *Load* završava sa izvršenjem. Ažuriranje arhitekturnog registra se ne obavlja sve dok se onstrukcija *Load* ne kompletira od strane bafera preuredjenja. Pri ovoj analizi usvajamo da se pristup memoriji za podatke od strane trećeg protočnog stepena obavlja u jednom mašinskom ciklusu. Ovakav scenario je moguć kada se koristi keš-podataka (U sekciji 4.3.2 dato je više detalja koji se odnose na keš). Kada je instaliran keš-podataka, može da se desi da podaci koji se pune u memoriju ne budu rezidentni u kešu za podatke. Ovo rezultira keš promašajem i zahteva da se keš-podataka napuni iz glavne memorije. Ovakvi tipovi keš promašaja dovode do zastoja *Load/Store* protočne jedinice.

Instrukcija *Store* se procesiraju na nešto drugačiji način u odnosu na instrukciju *Load* . Nasuprot instrukciji *Load*, za instrukciju *Store* smatramo da je završila sa svojim izvršenjem na kraju drugog protočnog stepena što odgovara aktivnosti uspešnog prevodjenja adrese. Registarski podatak koji se smešta u memoriju čuva se u baferu preuredjenja. Tek kada se instrukcija *Store* kompletira, podaci se upisuju u memoriju. Razlog za ovako zakašnjeni pristup memoriji predstavlja preuredjeno i potencijalno pogrešno ažuriranje memorije pa zbog toga instrukciju *Store* treba poništiti (*flush*), a do toga dolazi ako se javi izuzetak (*exception*) ili je predikcija instrukcije *Branch* pogrešna (*Branch missprediction*). S obzirom da se instrukcijom *Load* vrši čitanje memorije njeno poništavanje neće dovesti do nepoželjnih nus efekata (*side effects*) u stanju memorije. Kod instrukcije *Store*, umesto da se ažuriranje memorije vrši u fazi *completion* prvo se obavlja kopiranje (premeštanje) podataka u *Store* baferu. *Store* bafer je FIFO bafer koji baferuju arhitekturno kompletiranje instrukcije tipa *Store*. Svaka od ovih instrukcija tipa *Store* se zatim izvlači (*retire*), tj. u trenutku kada je memorijska magistrala dostupna ažurira se memorija. Svrha bafera *Store* je da obezbedi izvlačenje podataka koji se odnose na instrukciju *Store* u trenutku kada je memorijska magistrala slobodna, obezbedjujući pri tome prioritet instrukcijama tipa *Load* koje treba da pristupe memorijskoj magistrali. Termin kompletiranje (*completion*), ili dovršenje, se odnosi na ažuriranje stanja CPU-a, dok termin izvlačenje (*retiring*) se odnosi na ažuriranje memorijskog stanja. Kod bafera-*Store*, instrukcija *Store* je arhitekturno kompletirana ali njen sadržaj još nije izvučen (upisan) u memoriju. Kada se u toku izvršenja programa javi izuzetak, instrukcije koje u programskom redosledu slede nakon instrukcije čije je izvršenje dovelo do generisanja izuzetka, a koje su u medjuvremenu van-redosledno završile sa procesiranjem, moraju da se izbace (anulira njihov efekat) iz bafera preuredjenja.

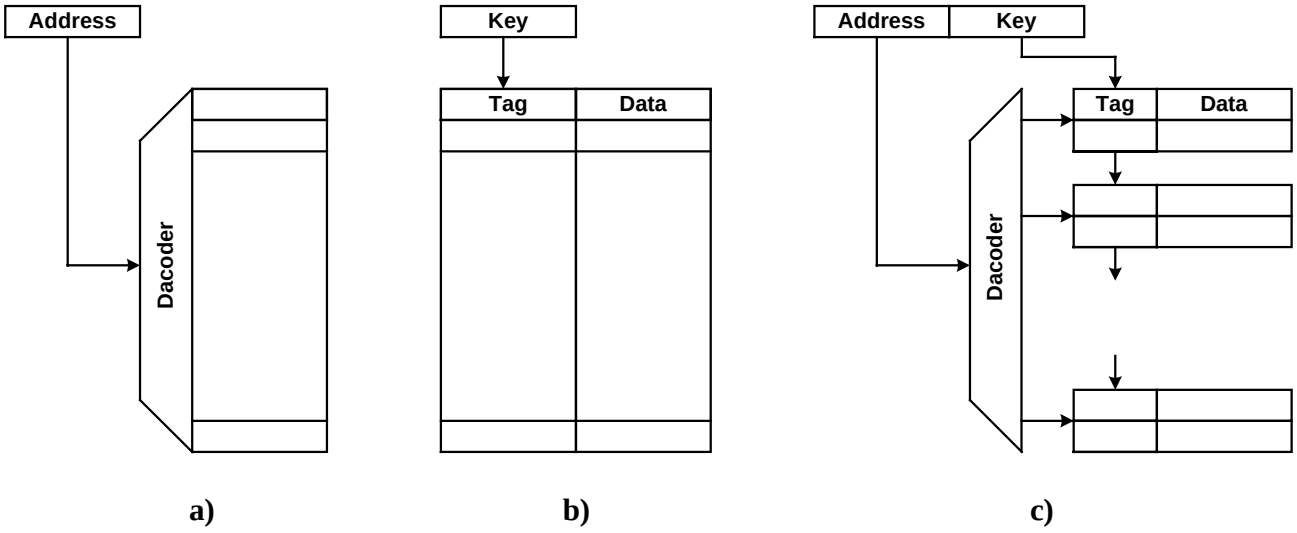
U toku sprovedene analize usvijili smo da se obe aktivnosti, prevodjenje adrese i pristup memoriji, obavljaju za jedan mašinski ciklus. Ovakva latentnost obično zahteva korišćenje *TLB*-ova i keševa. Jedan kratak pregled implementacija keš memorije i virtuelne memorije je dat u narednoj sekciji. Oni koji su familijarni sa konceptima hijerarhijske organizacije memorije mogu preskočiti ovu sekciju.

4.3.2. Hijerarhijska organizacija memorije

Pre nego što analiziramo neke specifične tehnike za prenos podataka ka/iz memorije neophodno je prvo sa implemenntacione perspektive da se podsetimo osnovnih principa rada keš i virtuelne memorije. Pri ovome posebnu pažnju abratićemo na sledeće četiri teme: mehanizmima za pristup memoriji, implementacijama keš memorije, *TLB* implementacijama, i interakciji izmedju keš memoriji i *TLB*-a.

Postoje sledeća dva osnovna načina za pristup memoiji sa većim brojem ulaza: a) indeksiranjem putem korišćenja adrese; i b) asocijativnim pretraživanjem preko korišćenja *tag*-a (adresnog markera).

Indeksirana-memorija koristi adresu da bi indeksirala memoriju sa ciljem da selektuje pojedini ulaz (vidi sliku 4.31a). Za dekodiranje *n*-to bitne adrese koristi se dekoder. Dekoder dozvoljava rad, radi upisa ili čitanja, jednom od 2^n ulaza. Postoji strogo odredjeno preslikavanje izmedju adrese i podataka, što u suštini znači da podatak mora biti upisan u memoriju u fiksan ulaz. Indeksirana-memorija je rigidna sa aspekta preslikavanja, ali nije kompleksna za implementaciju.



Slika 4.31 Mehanizmi za pristup memoriji (a) indeksirana memorija; (b) (potpuno) asocijativna memorija; (c) skupno-asocijativna memorija

Nasuprot indeksiranoj, asocijativna memorija u fazi pretraživanja memorije koristi ključ da bi selektovala pojedini ulaz (vidi sliku 4.31 b). Svakom memorijskom ulazu pridruženi su *tag* polje i komparator koji uporedjuje sadržaj svog *tag* polja i ključa. Ako dodje do uparivanja selektuje se taj ulaz. Korišćenje ovog oblika asocijativnog pretraživanja omogućava fleksibilno memorisanje podataka na bilo koju lokaciju u memoriji. Fleksibilnost se ostvaruje po ceni kompleksnije implementacije.

Kompromis izmedju indeksirane i asocijativne memorije predstavlja skupno-asocijativna memorija koja koristi kako indeksiranoo tako i asocijativno pretraživanje (vidi sliku 4.31 c). Adresa se koristi za indeksiranje na nivou jadnog od skupova, a pretraživanje većeg broja ulaza na nivou skupa obavlja se ključem čime se identifikuje pojedini ulaz. Ovaj kompromis obezbedjuje odredjenu fleksibilnost u smeštanju podataka bez da se pri tome ovo rešenje optereti kompleksnošću šeme koje je karakteristično za potpuno asocijativnu memoriju.

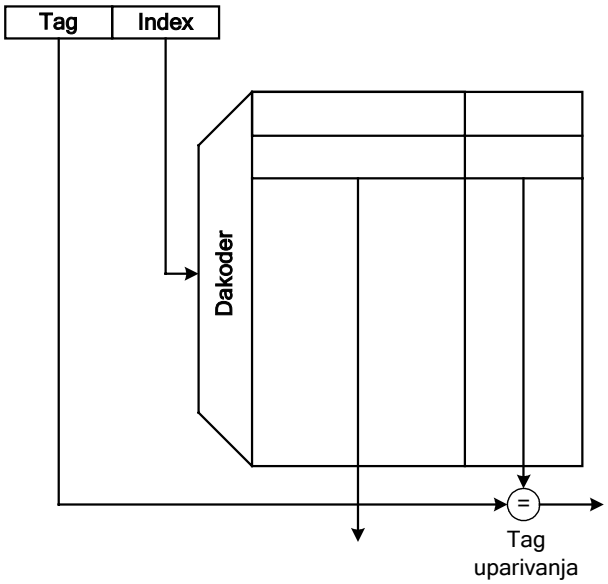
Memorijska hijerarhija se koristi da celokupnom memorijskom podsistemu obezbedi "iluziju" o velikom kapacitetu i niskom iznosu latentnosti. Niži nivoi memorije su obično veći, ali sporiji, dok su viši manji, ali brži. Oslanjajući se na latentnost kod obraćanja memoriji, sa ovakvom memorijskom

hijerarhijom mogu se ostvariti performanse koje su veoma bliske veoma brzim i velikim memorijama, bez da se pri tome poveća cena i kompleksnost. Keš memorija se postavlja izmedju memorije najvišeg nivoa i memorije nižeg nivoa kakva je glavna memorija.

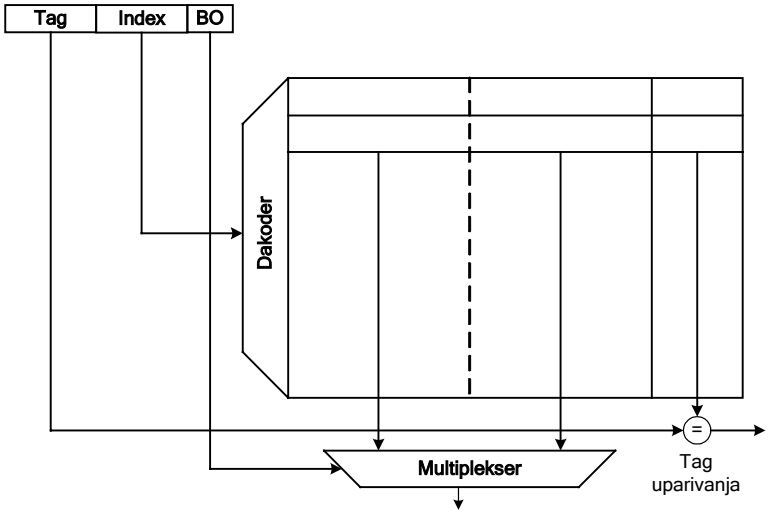
Glavna memorija se obično implementira kao deo velike indeksirane memorije. Sa druge strane, keš memorija se može implementirati korišćenjem jedne od tri šeme (slika 4.31) koje se odnose na pristup memoriji. Kada je keš memorija implementirana kao indeksna memorija, ona se naziva, vidi sliku 4.32, direktno-preslikani keš (*direct-mapped cache*). S obzirom da je direktno-preslikani keš memorija manjeg kapaciteta u odnosu na glavnu memoriju, njoj je potreban manji broj adresnih bitova pa njen manji dekoder može da dekodira podskup adresnih bitova glavne memorije. Saglasno tome, veliki broj adresa, kod direktno-preslikanog keša, se može preslikati na isti ulaz. Kako bi se obezbedili da selektovani ulaz sadrži korektne podatke, ostali, tj. nedekodirani, adresni bitovi moraju se koristiti radi identifikacije selektovanog ulaza. Zbog toga, pored polja podataka svakom ulazu je pridruženo *tag*polje koje se koristi za pamćenje nedekodiranih bitova. Kada je ulaz keša selektovan, pristupa se njegovom *tag* polju i uporedjuje se vrednost *tag* polja sa nedekodiranim bitovima originalne adrese kako bi se osigurali da taj ulaz sadrži podatke kojima se pristupa.

Na slici 4.32 a prikazan je direktno-preslikani keš kod koga svaki ulaz, ili blok, sadrži po jednu reč. Sa ciljem da se iskoristi prednost prostorne lokalnosti, blok keša može da sadrži veći broj reći (vidi sliku 4.32 b). Kada blok čine veći broj reči, neki od bitova originalne adrese moraju se iskoristiti za referenciranje pojeine reči u okviru bloka. To znači da se originalna adresa mora sada podeliti na tri dela: indeksni bitovi se koriste za selektovanje ulaza, blok ofset bitovi su namenjenji za selektovanje reči u okviru izabranog bloka, a *tag* bitovi se koriste za uparivanje sa *tag* bitovima zapamćenim u *tag* polju selektovanog ulaza.

Keš memorija se može takodje implementirati kao potpuno-asocijativna ili skupno-asocijativna memorija, vidi sliku 4.33 i sliku 4.34, respektivno. Potpuno-asocijativna memorija se karakteriše najvećom fleksibilnošću u odnosu na smeštanje podataka u ulazima keša. Osim blok ofset bitova, svi ostali bitovi se koriste kao ključ za potpuno-asocijativno pretraživanje svih ostalih ulaza keša. Potpuna asocijativnost obezbedjuje najefikasnije korišćenje svih ulaza keša, ali uslovljava najveću kompleksnost u implementaciji. Skupno-asocijativni keševi dozvoljavaju fleksibilni razmeštaj podataka izmedju svih ulaza skupa. Indeksni bitovi se koriste za selekciju pojedinog skupa, *tag* bitovi za selekciju ulaza u okviru skupa, a blok ofset bitovi za selekciju reči u okviru izabranog ulaza. Podela originalnih adresnih bitova na tri kategorije rezultat je pažljivog kompromisa.

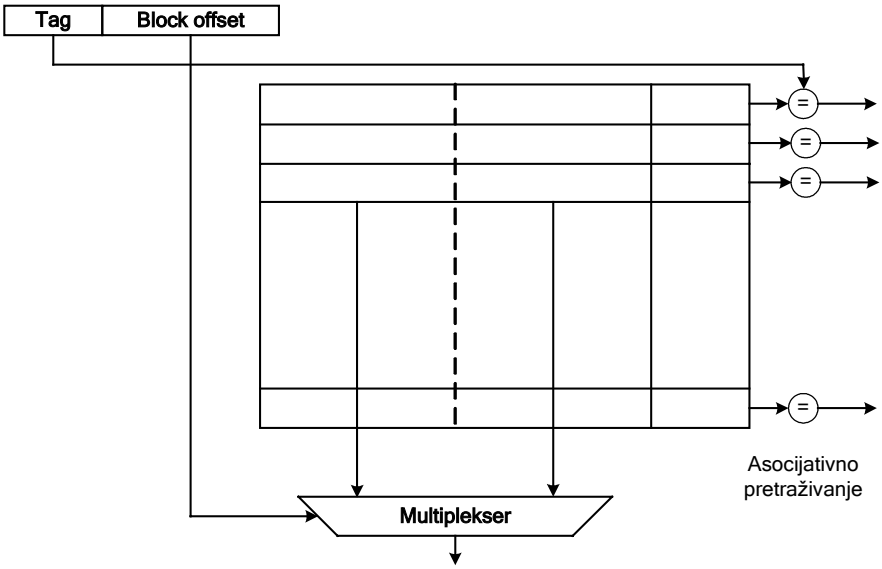


a)

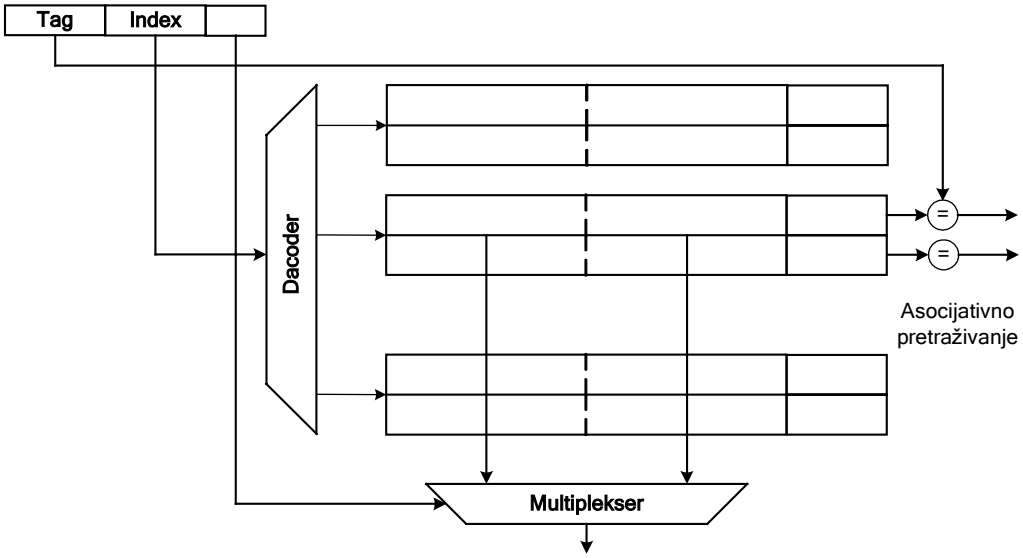


b)

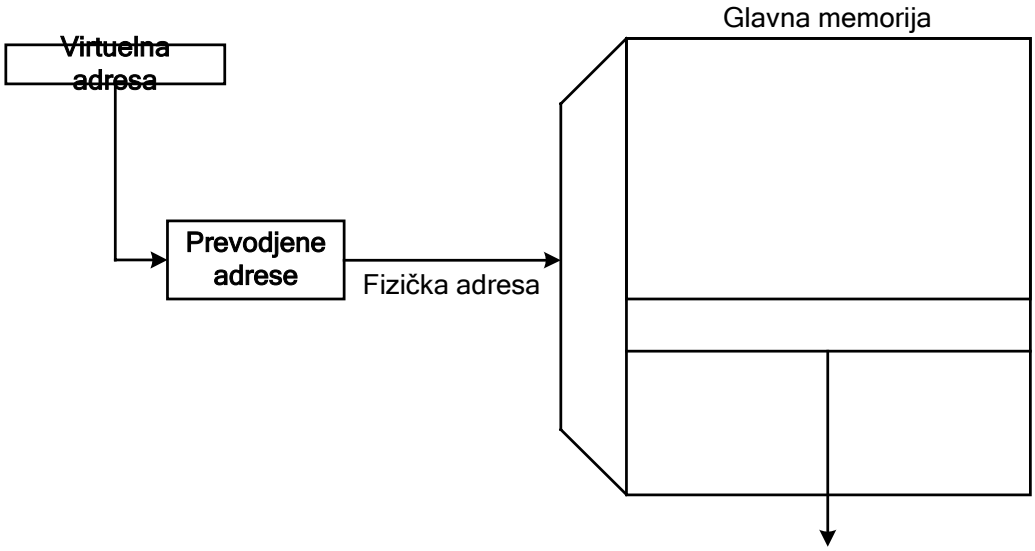
Slika 4-32 Direktno preslikani keševi: a) jedna reč po bloku; b) više reči po bloku



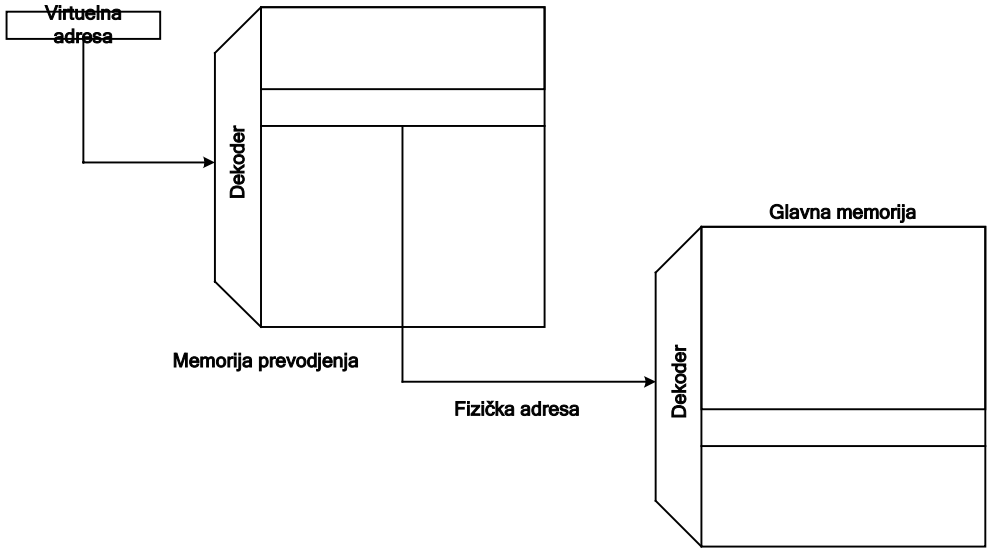
Slika 4-33 Potpuno-asocijativni keš



Slika 4.34 Skupno-asocijativni keš



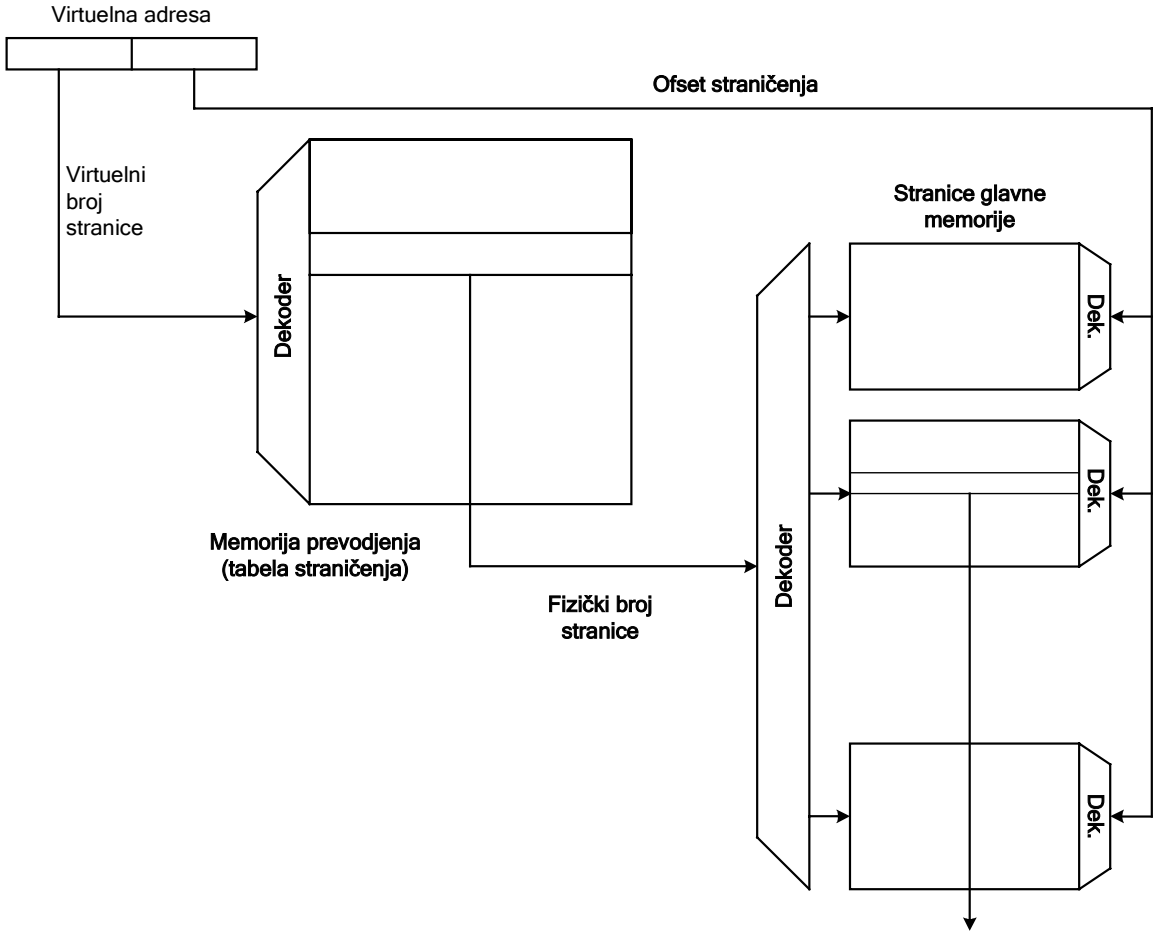
Slika 4-35 Prevodjenje virtuelne u fizičku adresu



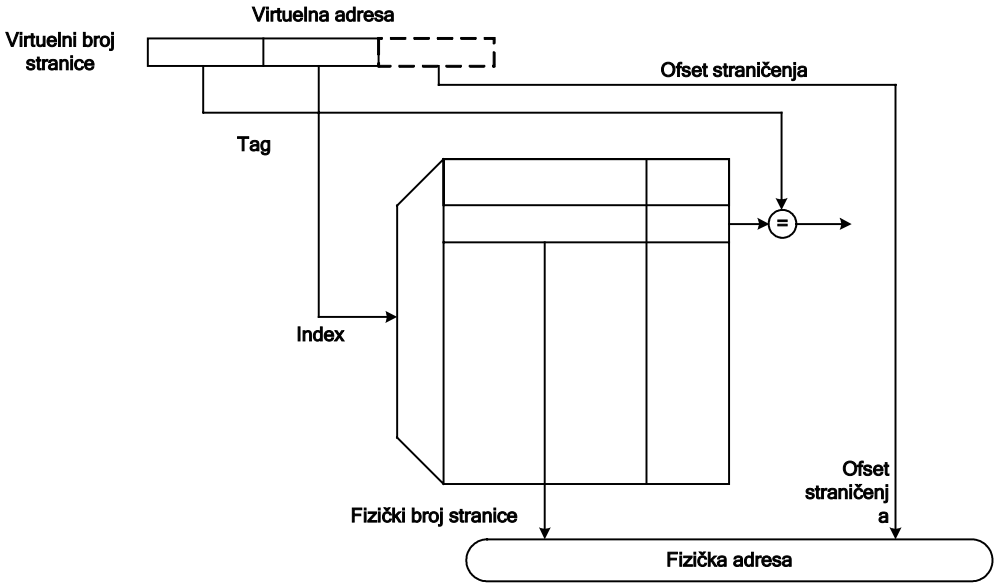
Slika 4.36 Prevodjenje virtuelne adrese reči u fizičku adresu reči koristeći memorije prevodjena (translacione memorije)

Zadatak virtuelne memorije je da obavi preslikavanje virtuelnog adresnog prostora u fizički adresni prostor. Ovaj zadatak uključuje prevodjenje virtuelne adrese u fizičku. Umesto da se direktno pristupa glavnoj memoriji pomoću adrese generisane od strane procesora, virtuelnu adresu koju generiše procesor prevodimo prvo u fizičku adresu. Fizička adresa, vidi sliku 4.35, koristi se zatim za pristup fizičkoj glavnoj memoriji. Prevodjenje adrese obavlja se pomoću translacione memorije. Pri ovome, virtuelna adresa se koristi kao indeks u translacionoj memoriji. Podaci koji se izbavljaju sa selektovanog ulaza translacione memorije koriste se kao fizička adresa za indeksiranje glavne memorije. To znači da fizičke adrese, koje odgovaraju virtualnim adresama, su zapamćene u odgovarajuće ulaze translacione memorije. Na slici 4.36 prikazan je način korišćenja translacione memorije u procesu prevodjenja adresa reči, tj. preslikavanja virtuelne adrese reči iz virtuelnog adresnog prostora u fizičku adresu reči u fizičkoj glavnoj memoriji.

Postoje dve slabosti koje se odnose na šemu translacione memorije prikazane na slici 4.36. Prvo, za prevodjenje adresa reči neophodna je translaciona memorija koja ima isti broj ulaza kao i glavna memorija. Ovakav pristup dovodi do dupliranja obima fizičke glavne memorije. U principu, da bi se ublažio problem, prevodjenje se može obaviti pomoću grublje granularnosti. Veći broj reči (obično stepena dva) u glavnoj memoriji se može zajedno grupisati u stranicu (*page*), pa je sada neophodno prevoditi samo adrese svake od stranica. U okviru stranice, selekcija reči se vrši pomoću LS bitova virtuelne adrese, tj. ofset bitovi stranice se direktno koriste bez potreba za prevodjenjem. Princip rada je prikazan na slici 4.37. Kod sistema koji koristi straničenje, translaciona memorija se naziva tabela straničenja (*page table*).



Slika 4.37 Prevodjenje virtuelne adrese stranice u fizičku adresu stranice korišćenjem memorije prevodjenja (translacione memorije)

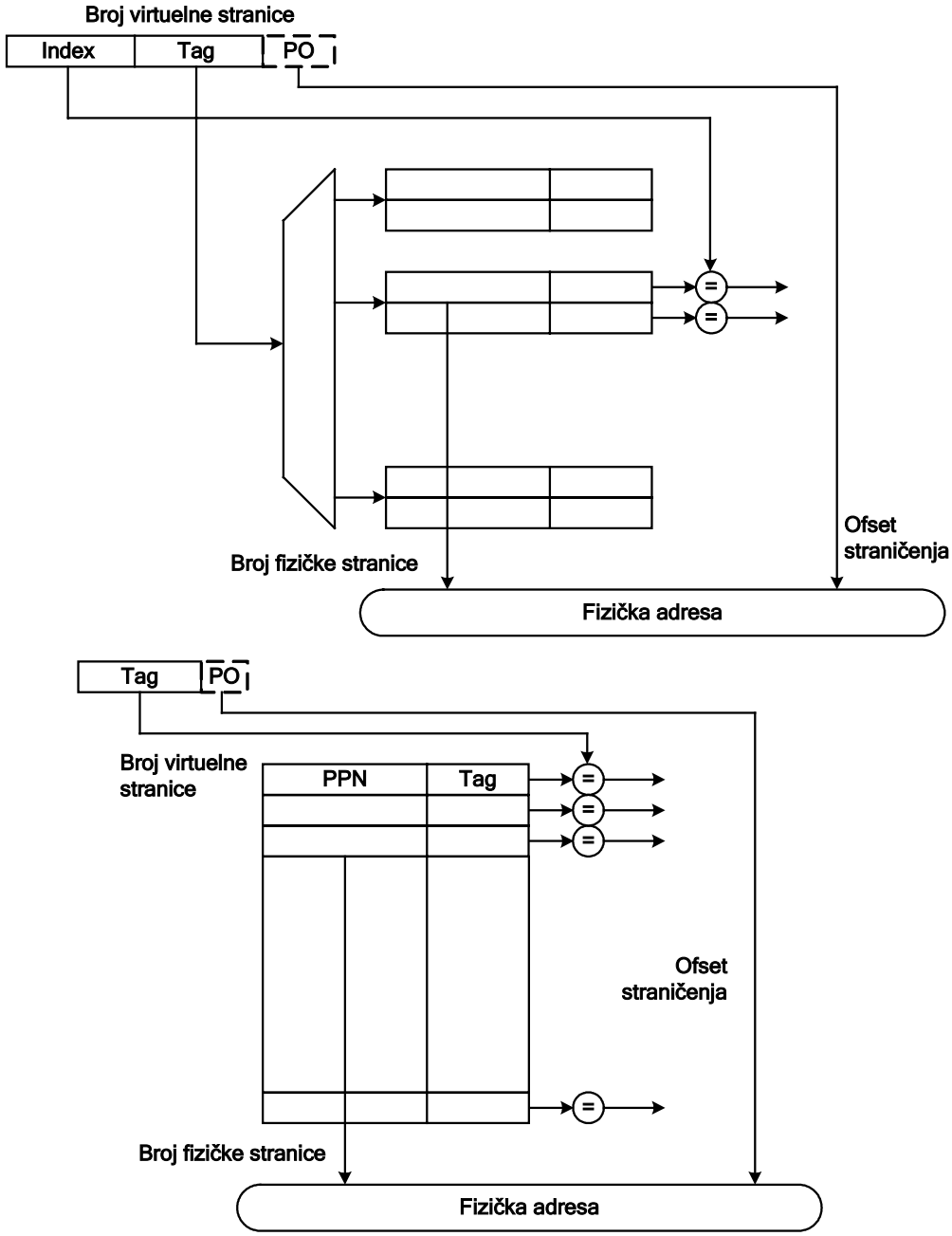


Slika 4.38 Direktno preslikani *TLB*

Druga slaba tačka translacione memorijske šeme ogleda se u činjenici da su sada svaki put kada se od strane instrukcije obraćamo glavnoj memoriji potrebna dva memorijska pristupa. Prvo, mora da se pristupi tabeli straničenja kako bi se dobavio broj fizičke stranice, a zatim se pristupa fizičkoj glavnoj memoriji koristeći broj prevedene fizičke stranice zajedno sa ofsetom stranice. Kod aktuelnih implementacija tabela straničenja se obično smešta u glavnu memoriju (obično u delu glavne memorije koji je lociran operativnom sistemu), pa shodno tome za svako referenciranje memorije od strane instrukcije potrebna su po dva sekvencijalna pristupa toj fizičkoj glavnoj memoriji. Ovakav način rada može postati usko grlo sa aspekta performansi. Rešenje se mora naći u implementaciji tabele straničenja koja koristi veoma brzu keš memoriju.

TLB (*Translation Lookaside Buffer*) je u suštini keš memorija za tabelu straničenja. Kao i svaka druga keš memorija, *TLB* se može implementirati korišćenjem bilo koje od tri šeme sa slike 4.31 koje se odnose na pristup memoriji. Direktno preslikani *TLB* je najmanja (i najbrža) verzija tabele straničenja. Virtuelni broj stranice deli se na indeks za *TLB* i *tag* (vidi sliku 4.38). Virtuelni broj stranice se prevodi u fizički broj stranice koji se povezuje sa ofsetom stranice da bi se formirala fizička adresa.

Sa ciljem da se ostvari malo fleksibilnije i efikasnije korišćenje *TLB* ulaza, *TLB*-ovoj implemetaciji se dodaje asocijativnost. Na slici 4.39 prikazani su skupno-asocijativni i potpuno-asocijativni *TLB*-ovi. Kod skupno-asocijativnog *TLB*-a, virtuelni i adresni bitovi se dele na tri polja: indeks, *tag*, i ofset stranice. Obim (veličina) polja koja se odnosi na ofset stranice diktiran je obimom stranice koji je specificiran od strane arhitekture i operativnog sistema. Ostala polja, tj. indeks i *tag*, čine broj virtuelne stranice. Kod potpuno-asocijativne *TLB*, indeks polja ne postoji, dok *tag* polje sadrži broj virtuelne stranice.

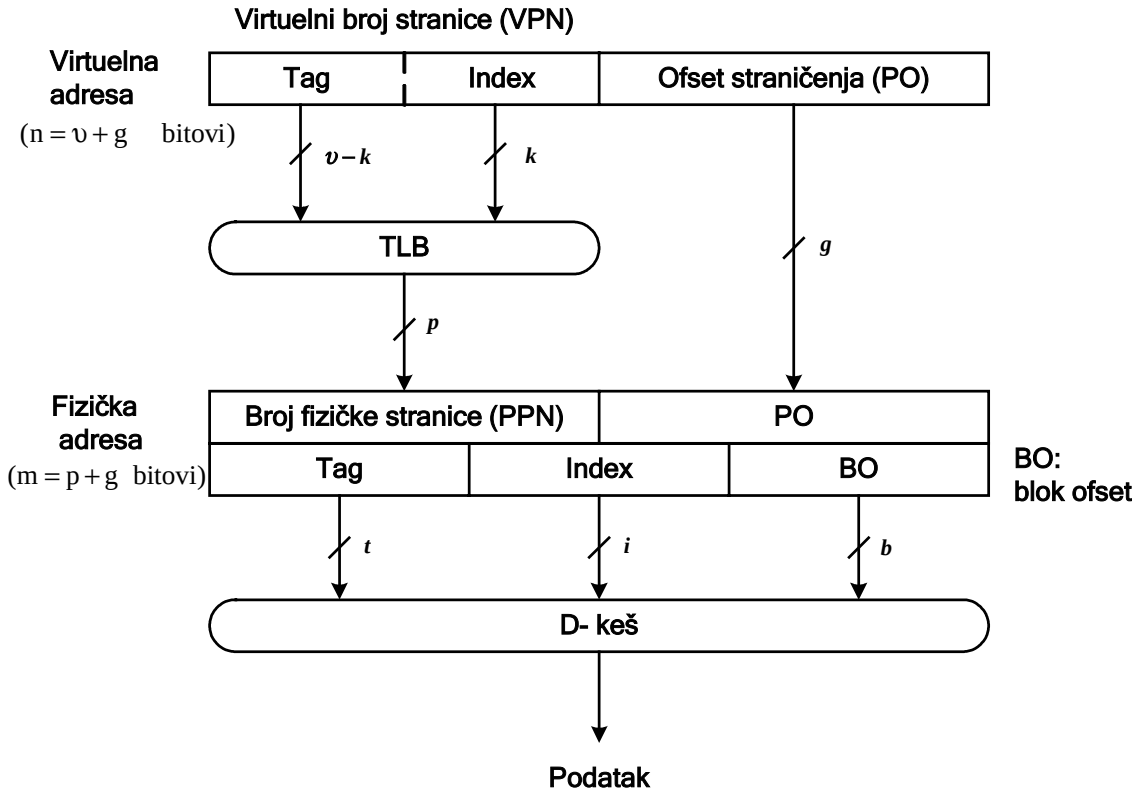


Slika 4.39 Asocijativni TLB-ovi: a) skupno-asocijativni TLB; b) potpuno-asocijativni TLB

Keširanje dela tabele straničenja u *TLB* obezbedjuje brzo adresno prevodjenje, ali ipak može doći do *TLB* promašaja (*TLB misses*). Istovremeno u *TLB*-u ne mogu biti prisutna sva preslikavanja tipa virtuelna stranica u fizičku stranicu. Kada se u toku pristupa iz tabele-straničenja koja se čuva u glavnoj memoriji. To može dovesti do većeg broja ciklusa tipa zastoj u radu protočnog sistema. Takodje je moguće da *TLB* promašaj dovede i do greške-straničenja (*page-fault*). Do greške-straničenja dolazi kada informacija o preslikavanju virtuelna stranica u fizičku stranicu ne postoji u tabeli straničenja. To znači da pojedina stranica koja se referencira nije rezidentna u glavnoj memoriji pa se zbog toga mora pribaviti iz sekundarne memorije. Opsluživanje greške-straničenja zahteva prizivanje operativnog sistema radi pristupa disk memoriji, a to potencijalno dovodi do pojave desetine hiljada ciklusa zastoja.

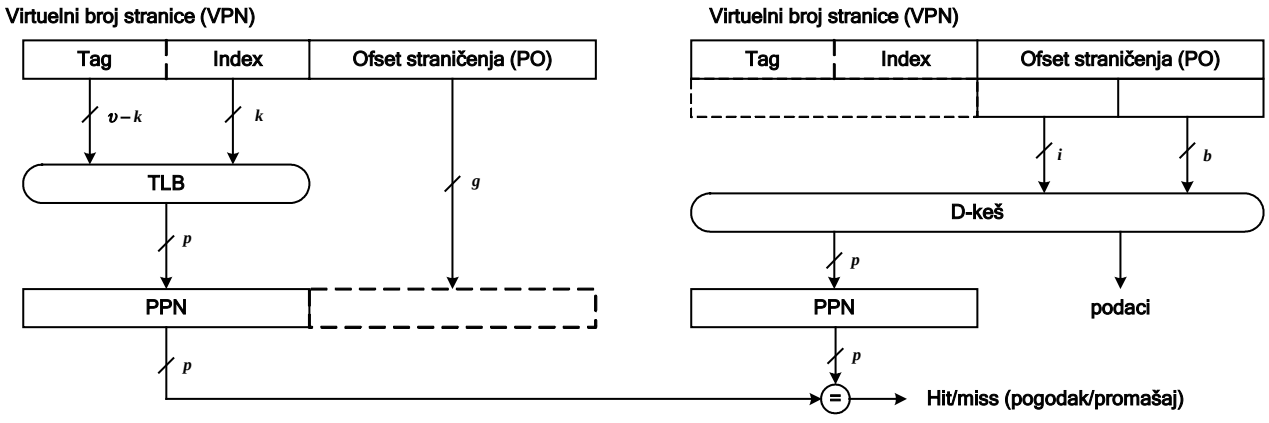
S toga, kada se od strane programa inicira greška-straničenja, izvršenje programa se suspenduje sve dok se od strane operativnog sistema ne opsluži greška-straničenja.

Keš podataka se koristi za keširanje dela glavne memorije, dok se *TLB* koristi za keširanje dela tabele-straničenja. Interakcija izmedju *TLB*-a i keša podataka je prikazana na slici 4.40. U odnosu na *Load/Store* protočnu jedinicu sa slike 4.30, *n*-bitna virtuelna adresa sa slike 4.40 predstavlja efektivnu adresu generisanu od strane prvog protočnog stepena. Virtelna adresa se sastoji od broja virtuelne jedinice (*v* bitova) i ofseta stranice (*g* bitova). Ako je *TLB* skupno-asocijativni keš, *v* bitova koji odgovaraju broju virtuelne stranice se dalje deli na *k*-to bitni indeks i (*v*-*k*)-to bitni *tag*. Drugi protočni stepen *Load/Store* jedinice koji je odgovoran za pristup *TLB*-u koristi virtuelni broj stranice. Ako pretpostavimo da ne postoji *TLB* promašaj, *TLB* će generisati fizički broj stranice (*p* bitova), koji se zatim povezuju sa *g*-bitova koji odgovaraju ofsetu stranice i generišu *m*-to bitnu fizičku adresu gde je $\boldsymbol{m = p + g}$, pri čemu je, u opštem slučaju, $\boldsymbol{m \neq n}$. U toku trećeg protočnog stepena pristup kešu podataka se vrši pomoću *m*-to bitne fizičke adrese. Ako blok keša podataka sadrži veći broj reči, tada se LS *b* bitova koristi kao ofset bloka radi selekcije referencirane reči iz selektovanog bloka. Selektovani blok se odredjuje na osnovu ostalih (*m*-*n*) bitova. Ako keš podataka skupno-asocijativni keš tada se ostalih (*m*-*n*) bitova deli na *t*-to bitni *tag* i *i*-to bitni indeks. Vrednost *i* se odredjuje na osnovu ukupnog obima keša i skupne asocijativnosti, tj. postoji *i*-skupova kod skupno-asocijativnog keša podataka. Kada nema keš promašaj, tada na kraju trećeg stepena (ako usvojimo da pristup kešu traje jedan ciklus) podatak biće dostupan na izlazu keša podataka (pod uslovom da se izvršava instrukcija *Load*).



Slika 4.40 Interakcija izmedju TLB-a i keša podataka

Organizacija prikazana na slici 4.40 ima nedostataka zbog toga što se *TLB*-u mora pristupiti pre pristupa kešu podataka. Serijalizacija pristupa *TLB*-u i kešu podataka unosi veliku latenciju koja odgovara zbiru dveju latencija. Ovo je razlog koji dovodi do toga da se na slici 4.30 prevodjenje adrese i pristup memoriji obavlja od strane dva različita protočna stepena. Rešenje ovog problema sastoji se u korišćenju virtuelno indeksiranog (*virtually indexed*) keša podataka koji omogućava paralelni pristup *TLB*-u i kešu podataka. Ova šema je prikazana na slici 4.41.



Slika 4.41 Virtuelno indeksirani keš podataka

Direktni način za implementaciju virtuelno indeksiranog keša podataka sastoji se u korišćenju ofset bitova stranice radi pristupa kešu podataka. S obzirom da bitovi za ofset u okviru stranice ne treba

prevoditi, oni se mogu koristiti bez prevodjenja. To znači da se *g* bitova ofseta stranice mogu koristiti kao blok ofset (*b*-bitova) i indeksa (*i*-to bitna) polja kada se pristupa kešu podataka. Radi pojednostavljenja, usvojićemo, da je keš podataka direktno-preslikani keš sa 2^i ulaza pri čemu svaki ulaz, ili blok, sadrži 2^b reči.

Umesto memorisanja ostalih bitova virtuelne adrese, tj. broj virtuelne stranice, kao njegovo *tag* polje, keš za podatke čuva prevedene fizičke stranice u svom *tag* polju. Ovo se obavlja u trenutku kada se popunjava linija keša podataka. U istom trenutku kada se bitovi ofseta stranice koriste za pristup kešu podataka, ostali bitovi virtuelne adrese, tj. broj virtuelne stranice, se koristi za pristup *TLB*-u. Ako usvojimo da su latencije pristupa *TLB*-u i kešu podataka izjednačene, u trenutku kada broj fizičke stranice *TLB*-a postane dostupan, *tag* polje (koje takodje sadrži broj fizičke stranice) keša podataka biće takodje dostupno. Dva *p*-to bitna broja koja odgovaraju fizičkoj stranici se uporedjuju sa ciljem da se odredi da li postoji pogodak (uparuju se brojevi fizičke stranice) u kešu podataka ili ne postoji. Kod virtuelno indeksiranog keša podataka, prevodjenje adrese i pristup kešu podataka se može preklopiti (*overlap*) sa ciljem da se smanji ukupna latencija.

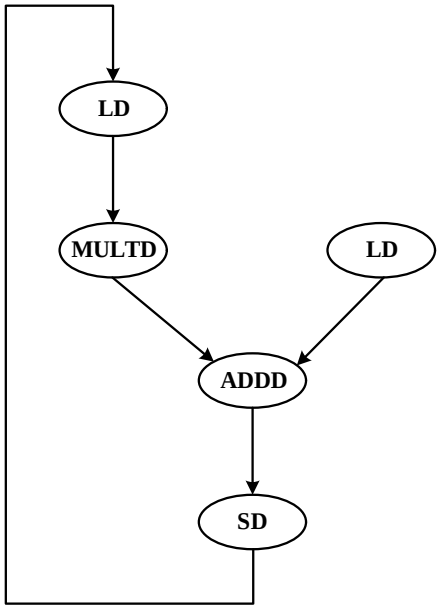
4.3.3. Uredjenje memorijskih pristupa

Memorijska zavisnost po podacima postoji izmedju dve *Load/Store* instrukcije ako obe instrukcije referenciraju istu memorijsku lokaciju, tj. postoji alijaza, ili kolizija, dveju memorijskih adresa. *Load* instrukcija obavlja čitanje memorijske lokacije, dok *Store* instrukcija obavlja upis u memorijsku lokaciju. Na sličan način registarske *zavisnosti po poadacima*, *RAW* ,*WAR* i *WAW* zavisnosti mogu takodje da postoje izmedju *Load* i *Store* instrukcija. *Store* (*Load*) instrukcija iza koje sledi *Load/Store* instrukcija podrazumeva obraćanje istoj memorijskoj lokaciji, a to dovodi do *RAW* (*WAR*) memorijske *zavisnosti po podacima*. Dve operacije *Store* na istu memorijsku lokaciju dovešće do pojave *WAW* zavisnosti. Sa ciljem da se očuva korektna semantika programa memorijske *zavisnosti po podacima* moraju biti korektno izvedene.

Jedan od načina da se izbegnu sve memorijske *zavisnosti po podacima* je da se sve *Load* i *Store* instrukcije izvrše u programskom redosledu. Ovakvo strogo uredjenje memorijskih instrukcija je dovoljeno, ali ne i neophodno da se izbegnu sve memorijske zavisnosti. Ono je konzervativno i uslovljava nepotrebna ograničenja koja smanjuju performanse programa. Koristićemo primer sa slike 4.42 da bi ukazali na ovaj aspekt. DAXPY je ime dela kôda koji se koristi za množenje vektora koeficijentom, a zatim sabiranje rezultantnog vektora sa drugim vektorom. DAXPY (izvedeno je od u duploj preciznosti pomnoži A sa X i saberi sa Y) predstavlja kernel kod LINPAC rutina koje se standardno koriste kod velikog broja numeričkih programa. Uočimo da su sve iteracije ove petlje nezavisne po podacima i da se mogu izvršavati paralelno. Ipak ako imamo u vidu ograničenje da se sve *Load/Store* ijstrukcije izvršavaju u pravom redosledu, tada prva *Load* instrukcija druge iteracije nemože da počne sve dok *Store* instrukcije prve iteracije ne završi. Ovo ograničenje će u suštini serijalizovati izvršenje svih iteracija u petlji.

```
  Y(i)=A * X(i) + Y(i)

     F0   LD,a
     R4   ADDI,Rx,#512       ;last address
Loop:
     F2   LD,0 (Rx)          ;load X(i)
     F2   MULD,F0,F2         ;A * X(i)
     F4   LD,0(Ry)           ;load Y(i)
     F4   ADDD,F2,F4         ;A * X(i) + Y(i)
     0(Ry)   SD,F4           ;store into Y(i)
     Rx   ADDI,Rx,#8         ;inc. index to X
     Ry   ADDI,Ry,#8         ;inc. index to Y
     R20   SUB,R4,Rx         ;compute bound
     BNZ,R20,Loop            ;check is done
```



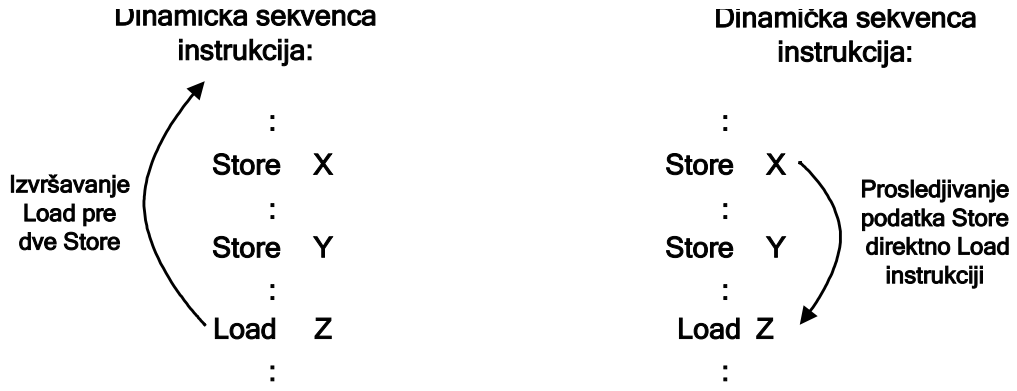
Slika 4.42 DAXPY primer

Ako dozvolimo da se *Load/Store* izvršavaju *van-redosleda* (*out-of-order*) bez narušavanja memorijskih *zavisnosti po podacima*, moguće je ostvariti poboljšanje performansi. Tako na primer ponovo analizirajmo primer DAXPY petlja. Graf na slici 4.42 prikazuje prave *zavisnosti po podacima* i uključuje ključne instrukcije tela petlje. Ove zavisnosti postoje izmedju instrukcija iste iteracije u petlji. Pri tome ne postoje *zavisnosti po podacima* izmedju većeg broja iteraciija u petlji. *Branch* instrukcija koja se nalatzi na kraju petlje je veoma prediktabilna, pa zbog toga pribavljanje iinstrukcije iz naredne iteracije se može obaviti veoma brzo. Isti arhitekturni registri koji se specificiraju od strane instrukcija iz narednih iteracija mogu se dinamički preimenovati od strane mehanizama za preimenovanje registara; pa stoga ne postoje registarske zavisnoti izmedju iteracija usled dinamičkog ponovnog korišćenja istih arhitekturnih registara. Saglasno tome ako je *Load/Store* instrukcijama bilo dozvoljeno da se izvršavaju *van-redosledno*, *Load* instrukcije iz potonje iteracije mogu da počnu pre izvršenja *Store* instrukcije iz prethodne iteracije. Preklapanjem izvršenja većeg broja iteracija u petlji rezultira povećanju performansi.

Memorijski modeli uvode odredjenja ograničenja koja se odnose na *van-redosledno* izvršenje *Load/Store* instrukcije od strane procesora. Kao prvo, da bi olakšali oporavak nakon izuzetaka, sekvencijalno stanje meorije mora biti očuvano. Drugim rečima stanje memorije mora da evoluira u saglasnosti sa sekvencijalnim izvršenjem *Load/Store* instrukcija. Kao drugo, kod najvećeg broja multiprocesorskih sistema usvaja se da postoji sekvencijalni konzistentni memorijski modul, a to zahteva da se pristup deljivoj memoriji od strane svakog procesora izvodi u saglasnosti sa programskim redosledom. Oba prethodno pomenuta razloga u suštini zahtevaju da se *Store* instrukcija izvršavaju u programskom redosledu, ili u najmanju ruku da se memorija mora ažurira kao da se memorisanja obavljaju u programskom redosledu. Ako je za memorisanje potrebno da se izvrašavaju u programskom redosledu, *WAW* i *WAR* memorijske *zavisnosti po podacima* implicitno su zakočene i ne inicira se njihovo izvršenje. Zbog toga samo *RAW* memorijske zavisnosti moraju biti zaustavljene.

4.3.4. Premošćavanje i prosledjivanje kod instrukcije *Load*

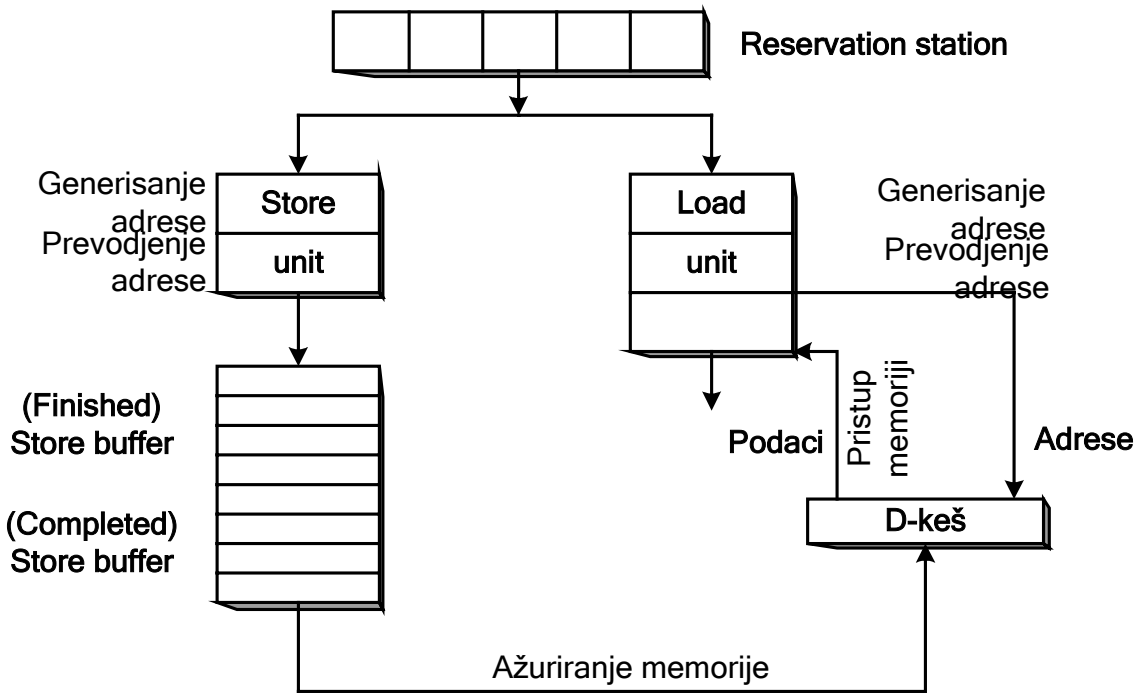
Van-redosledno izvršenje instrukcije *Load* instrukcija prestavlja primarno izvorište za postizanje boljih performansi. Kao što smo uočili kod DAXPY primera instrukcija *Load* se često javljaju na početku lanaca zavisnosti, tako da njihovo ranije izvršenje može da podstakne ranije izvršenje drugih zavisnih instrukcija. Relativno u odnosu na memorijske *zavisnosti po podacima*, *Load* instrukcije se mogu posmatrati kao operacije koje obavljaju čitanje memorijskih lokacija, a one takodje aktuelno obavljaju i operacije upis odredišne registare. Ako punjenja predstavljaju instrukcije za definisanje stanja registara (*DEF*), iza njih obično neposredno slede druge zavisne instrukcije koje koriste registar (*USE*). Cilj je da se obezbedi da *Load* instrukcije počnu sa izvršenjem što je moguće ranije, po mogućnosti da se nadju ispred drugih *Store* instrukcija koje im prethode, sve dok *RAW* memorijske *zavisnosti po podacima* nisu narušene i da se memorija ažurira u saglasnosti sa sekvencijalnim konzistentnim memorijskim modelom.



Slika 4.43 Ranije izvršavanje *Load* instrukcije: a) *Load* premošćavanje; b) *Load* prosledjivanje

Dve specifične tehnike za ranije *van-redosledno* izvršenje punjenja su *load bypassing* (punjenje sa premošćavanjem) i *load forwarding* (punjenje sa prosledjivanjem).Kao što je prikazano na slici 4.43 a) punjenje sa premošćavanjem obezbedjuje da se naredna *Load* izvrši ranije u odnosu na prethodnu *Store* pod uslovom da adresa *Load* instrukcije nije u alijazi sa prethodnom *Store*, tj. ne postoje *memorijske zavisnosti po podacima* izmedju *Store* i *Load* . Sa druge strane, ako izmedju potonje *Load* i naredne *Store* postoji alijaza, tj. postoji *RAW* zavisnost izmedju *Store* i *Load*, *load forwarding* omogućava da *Load* primi podatak direktno iz *Store* bez da se sačeka na pristup memoriji podataka.(vidi sliku 4.43b)). U oba prethodna slučaja postiže se ranije izvršenje *Load* instrukcije.

Pre nego što analiziramo činjenice koje moramo sagledati sa ciljem da implementiramo *load bypassing* i *load forwarding*, prvo ćemo predstaviti organizaciju dela *execution-jezgra* koje je odgovorno za procesiranje *Load/Store* instrukcija. Ova organizacija, prikazana na slici 4.44, koristiće se kao vodilja za dalju diskusiju koja se odnosi na *load bypassing* i *load forwarding*. Postoji samo jedna Store jedinica (organizovana kao dvo-protočni sistem) i jedna Load jedinica (organizovana kao tro-protočni sistem) pri čemu obe dobijaju podatke od strane jedinistvene rezervacione stanice. Za sada pretpostavićemo da se *Load* i *Store* instrukcije iniciraju radi izvršenje iz ove deljive rezervacione stanice u programskom redosledu. Store jedinica se podržava od strane Store bafera. Load jedinica i Store bafer mogu da pristupaju kešu podataka.



Slika 4.44 Mehanizmi za procesiranje *Load/Store*; posebne Load i Store jedinice sa redoslednim iniciranjem redosleda izvršenja instrukcija iz zajedničke rezervacione stanice

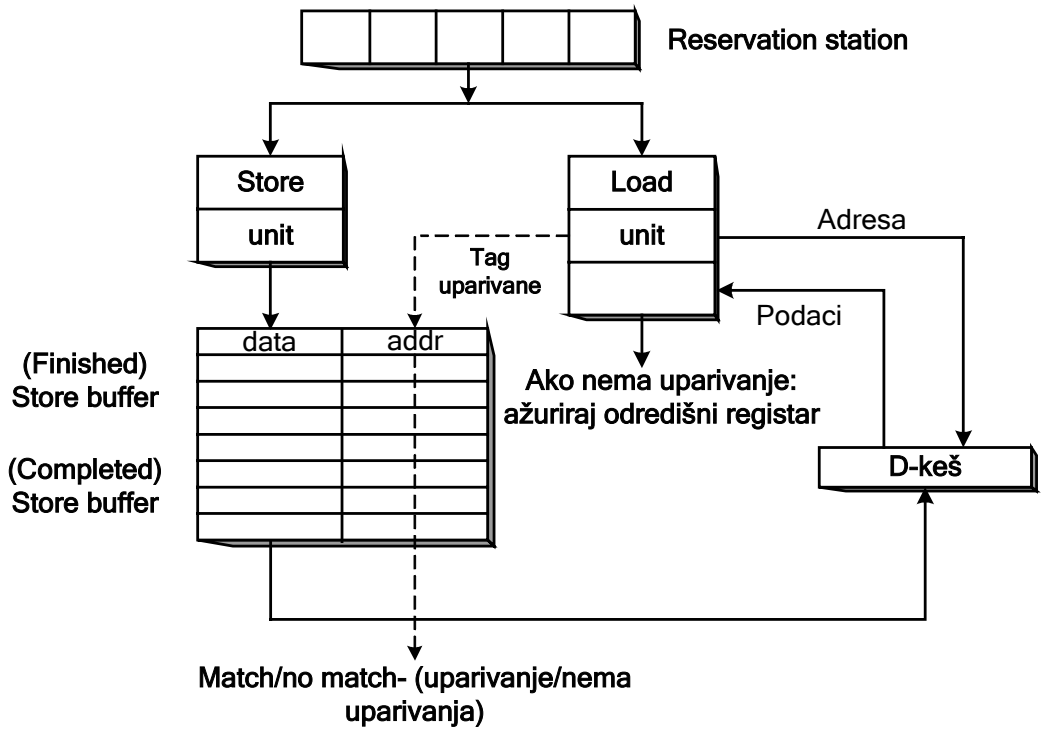
Za datu organizaciju sa slike 4.44, *Store* instrukcija, u toku procesiranja, se može naći u jednom od nekoliko stanja. Kada se *Store* instrukcija dispečuje u rezervacionu stanicu, alocira joj se ulaz u bafer preuredjenja. Ona ostaje u rezervacionoj stanici sve dok svi njeni izvršni operandi ne postanu dostupni i zatim se inicira u protočnu izvršnu jedinicu. Nakon što je memorijska adresa generisana i uspešno translirana (prevedena), smatra se da smo završili sa njenim izvršenjem i smešta se u deo Store bafera gde se smatra završenom (bafer preuredjenja se takodje ažurira). Store bafer radi kao red čekanja, a čine ga sledeće dve celine: *finished* i *completed*. Celina *finished* sadrži one *Store* koje su završile sa izvršenjem ali nisu još arhitekturno kompletirane. Celina *completed* bafera Store sadrži one *Store* koje su arhitekturno kompletirane ali čekaju na ažuriranje mamorije. Identifikacija ove dve celine Store bafera se obavlja preko pokazivača na Store bafer ili statusnog bita u ulazima Store bafera. *Store* u *finished* celinu Store bafera može potencijalno biti spekulativna, a kada se detektuje pogrešna spekulacija, ona treba da se isprazni iz Store bafera. Kada je *finished Store* kompletirana od strane bafera preuredjenja, ona se menja iz stanja *finished* u stanje *completed*. Ovo se izvodi ažuriranjem pokazivača Store bafera ili promenom statusnog bita. Kada kompletirana *Store* konačno napusti Store bafer i ažurira memoriju, za nju smatramo da je izvučena (*retired*). Posmatrano sa perspektive memorijske *Store*, u suštini *Store* ne završava sve dok nije izvučena. Ako se javi izuzetak, *Store* u kompletiranoj celini Store bafera mora da se propušti (zanemari njen efekat) kako bi se na adekvatan način ažurirala memorija. Na ovaj način izmedju dispečovanja i izvlačenja, za instrukciju *Store* kažemo da se može naći u jednom od sledeća tri stanja: *issued* (u Execution jedinici), *finished* (u *finished* celini Store bafera), ili *completed* (u *completed* celini Store bafera).

Jedna od ključnih stavki kod implementacije *load bypassing* sastoji se u potrebi za proverom radi moguće alijaze sa prethodnom *Store*, tj. one *Store* koje su premošćenje. Za *Load* smatramo da premošćava prethodnu *Store* ako *Load* čita iz memorije pre nego što *Store* upiše u memoriju. Stoga pre nego što je takvoj *Load* dozvoljeno da se izvrši ili pročita podatak iz memorije, neophodno je odrediti da li postoji alijaza sa svim prethodnim *Store* koje se nalaze u fazi procesiranja, tj. one čije je izvršenje

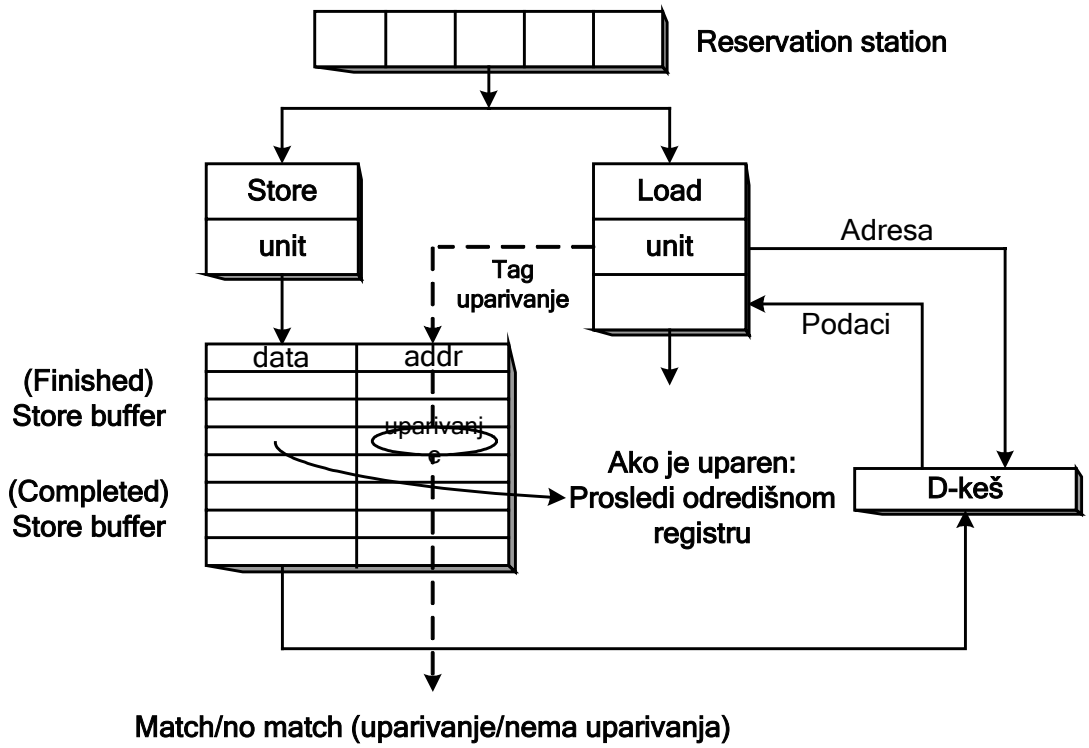
inicirano (*issued*), ali čiji rezultati još nisu izvučeni (*retired*). Ako usvojimo da, iz Load/Store rezervacione stanice, postoji redosledno iniciranje izvršenja *Load* i *Store* instrukcija, tada sve *Store* treba da budu smeštene u bafer-Store, uključujući kako *finished* tako i *completed* celine. Alijazom se proverava na sve moguće zavisnosti koje postoje izmedju *Load* i prethodne *Store*, pri čemu se provera ostvaruje korišćenjem Store bafera. U svaki ulaz Store bafera ugradjuje se *tag* polje koje sadrži memorijsku adresu. Nakon što je memorijska adresa *Load* instrukcije dostupna, ona se koristi da obavi asocijativno pretraživanje *tag* polja svih ulaza Store bafera. Ako dodje do uparivanja, tada za alijazu kažemo da postoji, a instrukciji *Load* nije dozvoljeno da se izvršava van-redosledno. U ostalim slučajevima, *Load* je nezavisna od prethodnih upisa u Store bafer i može se unapred izršavati. Asocijativno pretraživanje se obavlja u trećem protočnom stepenu Load jedinice konkuretnu sa pristupom kešu podataka; vidi sliku 4.45. Ako se ne detektuje alijaza instrukciji *Load* je dozvoljeno da završi, a odgovarajući preimenovani odredišni registar se ažurira sa podatkom koji se vraća iz keša podataka. Ako se alijaza detektuje podaci koji se vraćaju iz keša podataka se anuliraju a *Load* se zadržava u rezervacionoj stanici radi ponovnog budućeg iniciranja izvršenja.

Najveća kompleksnost u implementaciji *load bypassing*-a ogleda se u Store baferu i u pridruženom mehanizmu za asocijativno pretraživanje. Da bi se smanjila kompleksnost, *tag* polje koje se koristi za asocijativno pretraživanje može se redukovati tako da sadrži samo podskup adresnih bitova. Koristeći samo podskup adresnih bitova moguće je redukovati obim komparatora koji se koriste za asocijativno pretraživanje. Ipak, rezultat može biti pesimistički. Potencijalno alijaza može biti indicirana od strane ne tako obimnog komparatora u slučaju kada se ne kompariraju svi adresni bitovi. Neke od *load bypassing* mogućnosti se mogu izgubiti zbog egzistencije ovog kompromisa u implementaciji. U opštem slučaju degradacije performansi su minimalne ako se korsiti dovoljan broj bitova.

Tehnika za *load forwarding* može da poboljša i dopuni tehniku *load bypassing*. Kada je instrukciji *Load* dozvoljeno da preskoči prethodne *Store*, a da pri tome izmedju *Load* i prethodne *Store* postoji alijaza, tada se realno može ostvariti da *Load* koja direktno prosledjuje podatke koristi te podatke od *Store* sa kojom egzistira alijaza. U suštini, postoji memorijska *RAW* zavisnosti izmedju vodeće *Store* i prateće *Load* . Isto asocijativno pretraživanje Store bafera je u tom slučaju potrebno. Kada se detektuje alijaza, umesto da se sačuva *Load* tradi kasnijeg ponovnog iniciranja izvršenja, podatak sa ulaza koji ukazuje na alijazu, a pripada Store baferu, se prosledjuje preimenovanom odredišnom registru instrukcije *Load*. Ova tehnika ne samo da omogućava ranije izvršenje *Load* instrukcije, nego i eliminiše potrebu da *Load* pristupa kešu podataka. Na ovaj način se može povećati propusnost magistrale keša podataka.



Slika 4.45 Prikaz *load bypassing*(premošćavanja)



Slika 4.46 Prikaz *load forwarding* (prosledjivanja)

Da bi podržali *load forwarding* neophodno je usložniti Store bafer, vidi sliku 4.46. Kao prvo, svi adresni bitovi se moraju koristiti da bi se obavilo asocijativno pretraživanje. Kada se koristi

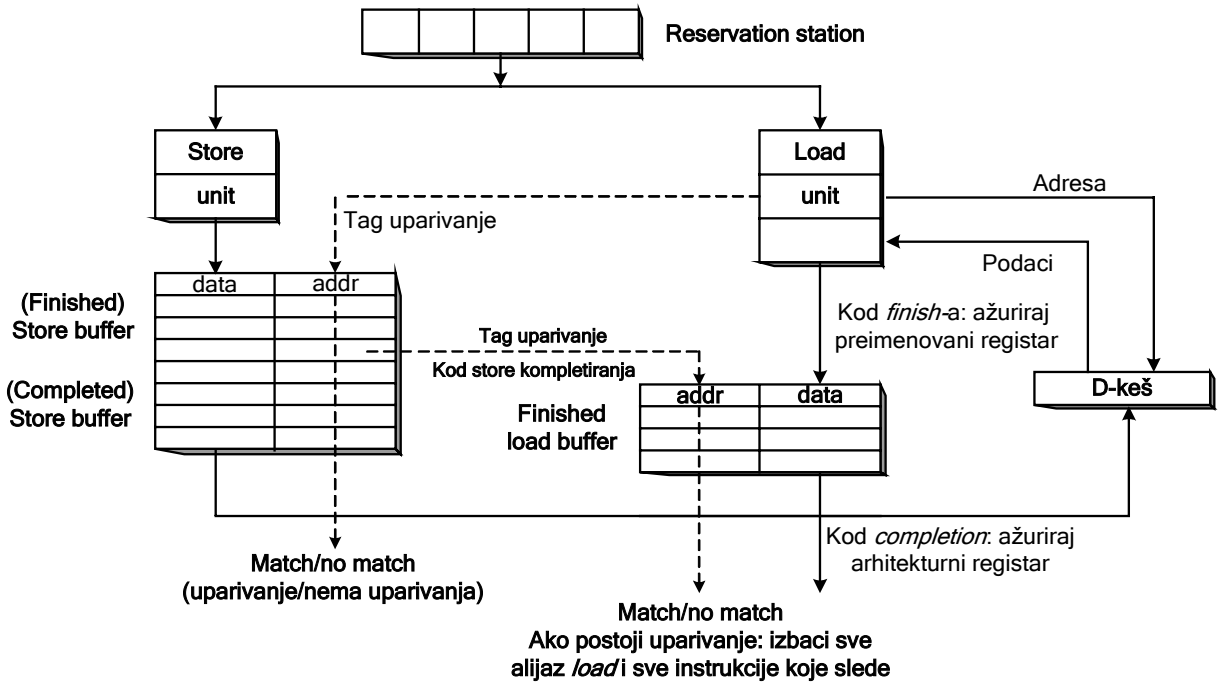
podskup adresnih bitova sa cilje da se podrži *load bypassing*, jedina negativna konsekvenca je gubitak mogućnosti. Kod *load forwarding* detekcija alijaze mora da se obavi pre nego što se obavi *forwarding* podataka, inače to će uzrokovati semantičku nekorektnost. Kao drugo, može da postoji veći broj prethodnih *Store* u Store baferu koji su u alijazi sa *Load*. Kada u toku asocijativnog pretraživanja dodje do većeg broja uparivanja, mora da postoji dodatna logika koja će odrediti koja od *Store* ima najskoriju alijazu. Ovo uzahteva ugradnju dodatne prioritetne logike kodiranja koja identifikuje najdavniju *Store* od koje je *Load* zavisna pre nego što se obavi *forwarding*. Kao treće, poterbno da se u Store bafer ugradi dodatni port za čitanje. Pre ugradnje *load forwarding*, store bafer je imao samo jedan port za upis preko koga je bio spregnut sa Store jedinicom i jedan port za čitanje preko koga je bio spregnut sa kešom za podatke. Novi port za čitanje koji se sada ugradjuje spreže se sa Load jedinicom; a sudari zbog pristupa portu se mogu javiti izmedju *load forwarding* i ažuriranja keša podataka.

Značajna performansa poboljšanja se mogu sotvariti pomoću *load bypassing* i *load forwarding*. Po *Mike Johnson*-u obično *load bypassing* rezultira povećanju performansi od 11 % - 19 %, a *load forwarding* dodatnom poboljšanju od 1 % - 4 %.

U toku dosadašnje analize usvojili smo da *Load* i *Store* dele istu rezervacionu stanicu sa instrukcijama čije se izvršenje inicira, u programskom redosledu, iz rezervacione stanice ka Store i Load jedinicama. To znači da pretpostavka o redoslednom iniciranju izvršenja instrukcija polazi od toga da će sve prethodne *Store* i *Load*, u trenutku kada se *Load* izvršava, nalaziti u store baferu. Na ovaj način se pojednostavljaje provera memorijske zavisnosti, pa kažemo da je neophodno obaviti samo asocijativno pretraživanje store bafera. Ipak pretpostavka o redoslednom iniciranju izvršenja instrukcija unosi nepotrebna ograničenja kod *van-redoslednog* izvršenja *Load*. Instrukcija *Load* može da bude spremna za iniciranje izvršenja, ali pri tome, prethodna *Store* može da zadržava iniciranje izvršenja instrukcije *Load* i pored toga što izmedju ove dve memorijske instrukcije ne postoji alijaza. To znači da dozvoljavanje van-redoslednog iniciranja izvršenja *Load* i *Store* instrukcija iz Load/Store rezervacione stanice može da obezbedi veći stepen van-redoslednog izvršenja prethodnih *Load* instrukcija. Ovo je posebno povojno u slučajevima kada se *Load* nalaze na početku kritičnih lanaca zavisnosti tako da njihovo ranije izvršenje može da otkloni kritična performansna uska grla.

Ako je *van-redosledno* iniciranje izvršenja instruckija dozvoljeno iz Load/Store rezervacione stanice, tada je potrebno rešiti jedan dodatni problem. Ako je *Load* instruikciji dozvoljeno iniciranje izvršenja van-redosleda, tada je moguće za neke od *Store* instrukcija koje joj prethode da se nalaze još u rezervacionoj stanici ili da budu u izvršnim protočnim stepenima, a ne u store baferu. Stoga, jednostavno obavljanje asocijativnog pretraživanja ulaza store bafera nije adekvatno kad se proverava potencijalna alijaza izmedju *Load* i sve prethodne *Store* instrukcija. Što je još gore, memorijske adrese prethodnih *Store* koje se i dalje nalaze u rezervacionoj stanici ili u izvršnim protočnim stepenima mogu biti još nedostupni.

Jedan od prstupa je da se obezbedi instrukciji *Load* da nastavi, usvajajući da ne postoji alijaza sa rethodnim *Store* koje se još ne nalaze u store baferu, a zatim da se kasnije obavi njihova validacija. Kod ovakvog pritstupa, instrukciji *Load* je dozvoljeno da se inicira radi izvršenja van.redosleda i da se izvršava spekulativno. Dakle, ako izmedju nje i bilo koje *Strore* u store baferu ne postoji alijaza , inistrukciji *Load* je dozvoljeno da završi sa izvršenjem. Ipak ovaj *Load* mora da se sačuva u novi bafer koji se naziva *finished load* bafer (vidi sliku 4.47). *Finished load* baferom se upravlja na sličan način kao i *finished store* baferom. *Load* je jedino rezidentan u *finished load* bafer nakon što ona završi sa izvršenjem, ali pre nego što se kompletira. Uvek kada se *Store* instrukcija kompletira, ona mora da obavi proveru radi alijaze u odnosu na inatrukcije *Load* koje senalaze u *finished load* baferu. Ako se ne detektuje alijaza, instrukciji *Store* je dozvoljeno kompletiranje. Ako se detektuje alijaza, tada to znači da postoji prateća instrukcija *Load* koja je zavisna od *Store*, i da je *Load* već završila sa izvršenjem. To znači da spekulativno izvr šenje te instruckije *Load* mora da se invalidoira i koriguje ponovnim iniciranjem izvršenja instruckije, ili čak ponovnim pribavljanjem, te instrukcije *Load* kao i sve naredne instrukcije. Ovo može da iziskuje uznačajnu hardversku kompleksnost i visoku cenu koja se plaća zbog degradacije performansi.



Slika 4.47 Potpuno *van-redosledno* iniciranje izvršenja kao i izvršenje *Load* i *Store* instrukcija

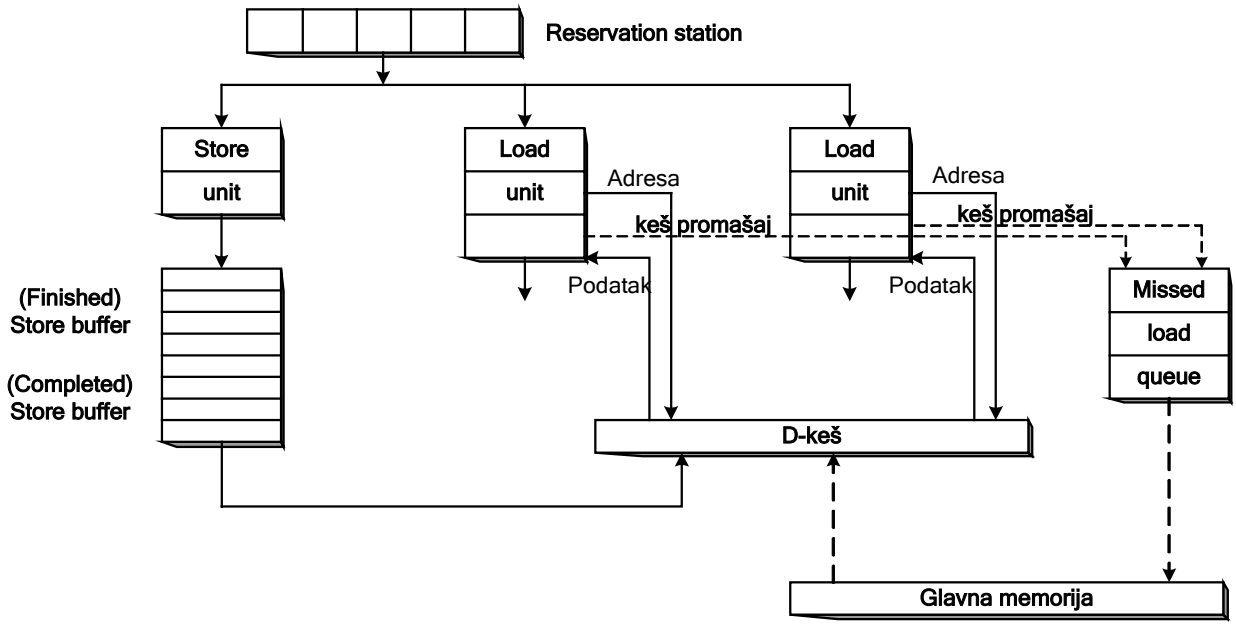
Agresivno ranije iniciranje izvršenja *Load* instrukcija može da dovede do značajnog povećanja performansi. Mogućnost da se spekulativno inicira izvršenje *Load* ispred *Store* može da dovede do ranijeg izvršenja većeg brohja uzavisnih instrukcija od kojih neke mogu biti različite od *Load*. Ovo je posebno važno za slučaj kada se javljaju keš promašaji. Ranije iniicranje izvršenja instrukcija *Load* može da dvovede do ranije pojave keš promašaja što dovodi do maskiranja nekih ili svih ciklusa koji se javljaju zbog keš promašaja. Negativna strana spekulativnog iniciranja izvršenja instrukcija *Load* predstavlja potencijalno režijsko vreme koje se odnosi na oporavljanje od pogrešne spekulacije. Jedan od načina da se smanji ovo režijsko vreme je da se ostvari alijaza ili zavisna predikcija. Kod tipičnih programa odnos zavisnosti izmedju *Load* i njene prethodne *Store* instrukcije je veoma predvidljiv. Prediktor memorijske zavisnosti se može implementirati sa ciljem da predvidi da li će *Load* imati alijazu sa prethodnim *Store*. Ovakav prediktor se može koristiti da bi odredio kada treba, ili ne treba spekulativno inicirati izvršenje *Load* instrukcija. Da bi dobili stvarno performanso poboljšanje sukcesivna spekulativna iniciranja *Load* instrukcija moraju se obavljati veoma razumno.

4.3.5. Druge tehnike za memorijski protok

Pored *laod bypassing* i *load forwarding*, postoje i druge tehnike za memorijski protok. Sve ove tehnike imaju za cilj da povećaju memorijsku propusnost i/ili redukuju memorijsku latentnost. Kako superskalarni procesori postaju obimniji, veća memorijska propusnost koja je u stanju da podrži veći broj *Load/Store* instrukcija po ciklusu je takodje potrebna. Zbog dispariteta zbog brzine procesora sa jedne strane i brzine memorije sa druge strane povećava se latentnost pruistupa memoriji, tako da posebno kada se javi keš promašaj dolazi do pojave uskog grla u performansama mašine.

Jedan od načina da se poveća memorijska propusnost je da se koristi veći broj Load/Store jedinica u *execution* jezrgu, čiji je rad podržan od strane multi-portnog keša podataka. U sekciji 4.3.4 mi smo usvojili da postoji jedna store jedinica i jedna load jedinica čiji je rad podržan od strane jedno-portnog keša podataka. Load jedinica ima priritet kod pristupa kešu podataka. *Store* instrukcije se smeštaju u redu čekanja store bafera i izvlače se iz njega ka kešu podataka onog trenutka kada

memorijska magistrala nije zauzeta a store bafer stekne pravo upravljanja nad kešom za podatke. Ukupna propusnost memorije podataka ograničena je na jednu *Load/Store* instrukciju po ciklusu. Ovo pretstavlja ozbiljno ograničenje posebno kada dodje do pojave paketnih (*burst*) *Load* instrukcija. Jedan od načina da se premosti ovo usko grlo je da se ugrade dve Load jedinice,kako je to prikazano na slici 4.48, kao i dvo-portni keš podataka. Dvo-portni keš podataka treba da je u stanju da podrži dva simultana prsitupa kešu u svakom ciklusu. Na ovaj način udvostručava se potencijal memorijske propusnosti. Ipak to se ostvaruje po ceni povećanje kompleksnosti hardvera, jer dvo-portni keš zahteva udvostručavanje hardvera keša. Jedan od načina da se premosti ova povećana cena hardvera sastoji se u implementaciji *interleaved* (preklapajućih) banki keša podataka. Kada se keš podataka implementira pomoću većeg broja memorijskih banaka, dva simultana pristupa različitim bankama mogu biti podržane u jednom ciklusu. Ako dva pristupa treba da pritupaju istoj banki javlja se konflikt zbog korišćenja banaka, tako da se oba pristupa moraju serijalizovati. Sa praktične tačke gledišta, keš koga čine 8 banaka može da očuva učestanost konflikta kod pristupanja bankama na prihvatljivi nizak nivo.



Slika 4.48 Dvo-portni ne blokirajući keš podataka

Napomena:: *Missed Load queue*- red čekanja promašenih *Load* instrukcija

Najstandardniji način da se smanji memorijska latencija ostvaruje se korišćenjem keša. Keš memorije se danas masovno koriste. Kako se razlika u brzini pristupa izmedju procesora i memoriji povećava, neophodno je koristiti veći broj nivoa keša memorije. Kod najvećeg broja visoko-performansnih superskalarni procesora inkorporirana su najmanje dva nivoa keša. Prvi nivo keša (L1) ima vreme pristupa od jednog ili nekoliko ciklusa. Obično postoje popsebni L1 keševi za čuvanje instrukcija i podataka. Drugi nivo keša (L2) obično podržava memorisanje kako instrukcije tako i podataka, a može biti ugradjen u čipu (*on-chip cache*) ili van čipa (*off-chip cache*). Ovoj memoriji se može pristupati serijski, u slučaju da postoji promašaj na nivou L1, ili paralelno kada se istovremeno pristupa i kešu L1. Kod nekih rešenja koristi se i nivo tri (L3). Predvidjana su da će u bliskoj budućnosti L3 keš postati standardno rešenje. Pored korišćenja keša i hijerarhijske organizacije keševa postoje i još dve druge tehnike za smanjenje efektivne memorijske latentnosti koje se nazivaju *nonblocking cache* i *prefetching cache*.

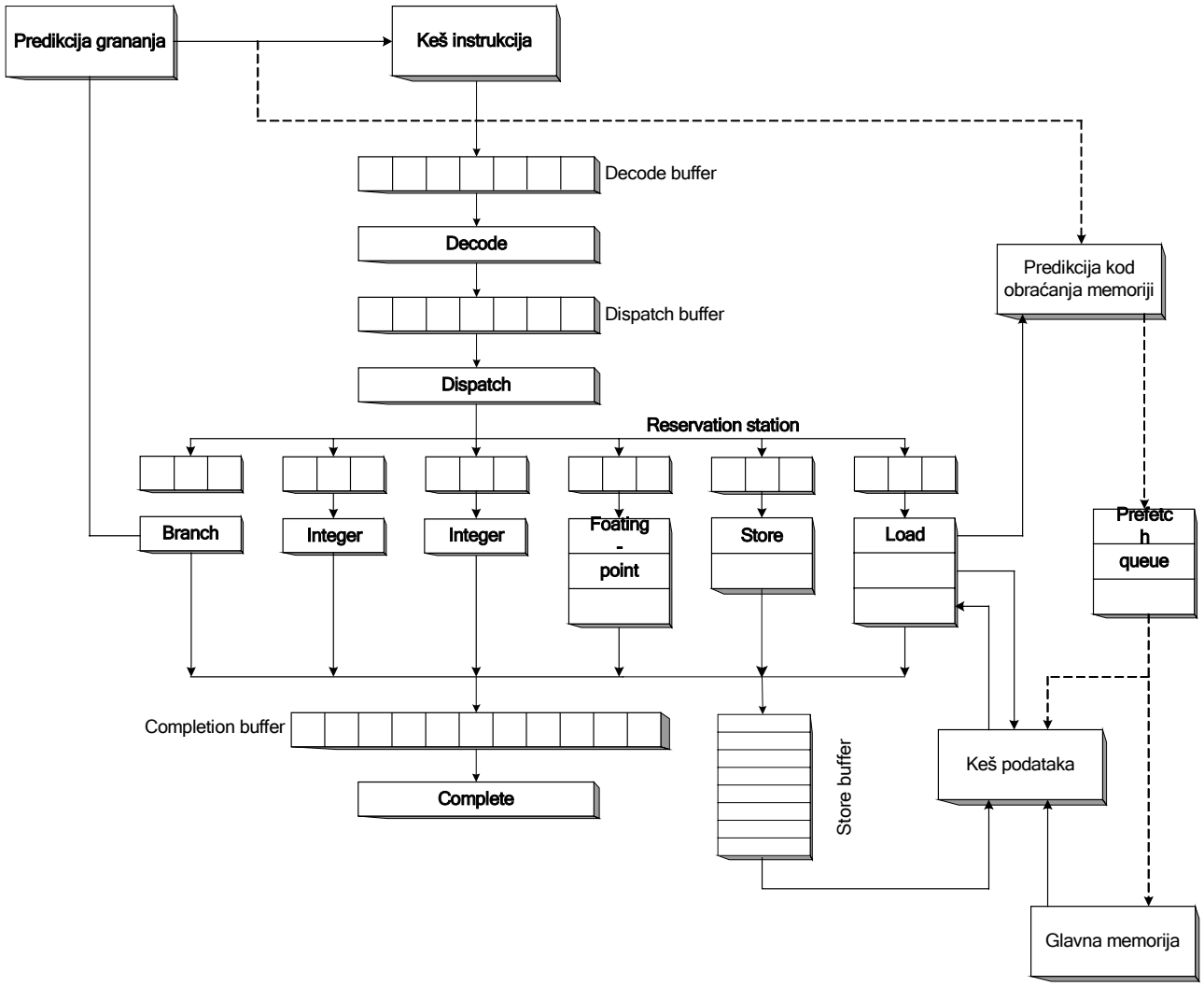
Nonblocking keš može smanjiti efektivnu memorijsku latenciju smanjenjem broja keš promašaja. Obično, kada se u toku *Load* instrukcije javi kerš promašaj, on će zaustaviti rad Load

protočne jedinice a shodno tome i dalje iniciranje izvršenja *Load* instrukcija sve dok se keš promašaj ne opsluži. Ovakav oblik zastoja je sa aspekta preklapanja konzervativan i brani iniciranje izvršenja narednih i nezavisnih *Load* instrukcija za koje se može javiti pogodak u kešu podataka. Ne blokirajući keš podataka premošćava ovaj nedostatak postavljanjem na stranu *Load* instrukcije koja je bila uzrok keš promašaja u specijalan red čekanja *Load* instrukcija zbog promašaja (*missed load queue*) i obezbedjujući iniciranje izvršenja narednim *Load* instrukcijama (vidi sliku 4.48). *Load* instrukcija zbog koje se javio promašaj smešta se u *missed load queue* sve dok se keš promašaj ne opsluži. Kada se blok zbog koga se javio promašaj pribavi iz glavne memorije, *Load* instrukcija zbog koje se javio promašaj izlazi iz *missed load queue* i završava sa izvršenjem.

U suštini ciklusi koji se javljaju usled keš promašaja se preklapaju, i maskiraju, procesiranjem narednih nezavisnih instrukcija. Naravno, ako naredne instrukcioje zavise od instrukcije *Load* zbog kooje se javio promašaj, iniciranje izvršenja te instrukcije se zaustavlja. Broj ciklusa u toku kojih dolazi do zastoja se može maskirati o zavisi od broja nezavisnih instrukcija koje slede nakon instrukcije *Load* u toku koje se javio promašaj. *Missed load queue* sadrži veći broj ulaza koji omogućavaju konkuretno opsluživanje većeg broja *Load* instrukcija u toku kojih se javio promašaj. Potencijalno ciklusi zastoja u radu keša zbog većeg broja *Load* instrukcija u toku kojih se javio promašaj mogu se preklapati što rezultira manjim brojem ciklusa zastoja.

Broj iniciranja izvršenja mora se takodje razmatrati kada se implementiraju neblokirajući keševi. *Load* promašaji mogu da se jave u paketima (*burst*). mogućnost da se podržava veći broj promašaja i obavi preklapanje njihovog opsluživanja je važna. Interfejs prema glavnoj memoriji, ili niži nivo keša, mora biti u stanju da podrži preklapanje ili protočno izvršenje većeg broja pristupa. Popunjenost keša trigerovan od strane instrukcije *Load* u toku koje se javio promašaj može da dodje u konflikt sa store baferom usled porta za upis u taj keš. Postoji jedna komplikacija koja može da iskrsne kod ne blokirajućih keševa. Ako je *Load* instrukcija zbog koje se javio promašaj na spekulativnom putu, tj. prediktovanom putu tada postoji verovatnoća da će spekulacija, tj. predikcija grananja, biti nekorektna. Ako je ova *Load* na pogrešno prediktovanom putu pitanje je kada keš promašaj treba da se opsluži. Kod mašina koje koriste agresivnu predikciju grananja broj instrukcija *Load* na pogrešno prediktovanom putu može biti značajan, tako da njihovi promašaji spekulativno mogu da zahtevaju značajnu memorijsku propusnost. studije pokazuju da ne blokirajući keš može da smanji cenu zbog *Load* promašaja na oko 15 %.

Drugi način da se smanji ili maskira cena koja se plaća zbog keš promašaja sastoji se u korišćenju *prefetching* keša. *Prefetching* keš sudeluje u buduće keš promašaje i ranije trigeruje keš promašaje tako da dolazi do preklapanja promašaja sa procesiranjem instrukcija koje prethode keš promašaju. Na slici 4.49 prikazan je *prefetching* keš podataka. Za implementaciju *prefetching* keša potrebne su dve strukture: *memory references predictione table-MRPT* (memorijska tabela za predikciju obraćanja) i *prefetching queue-PQ* (red čekanja unapred pribavljenih instrukcija). *MRPT* čuva insformaciju o prethodno izvršenim instrukcijama *Load* u tri različita polja. Prvo polje sadrži adresu instrukcije *Load* i koristi se kao *tag* marker za selekciju ulaza u tabelu. Drugo polje sadrži adresu prethodnog podatka instrukcije *Load*, dok treće sadrži vrednost koja ukazuje na razliku izmedju prethodne dve adrese podataka koje se koriste od strane te *Load*. *MRPT*-u se pristupa pomoću asocijativnog pretraživanja koristeći *fetch* adresu generisanu od strane prediktora grananja i prvog polja tabele. Kada postoji *tag* uparivanje, koje ukazuje na pogodak kod *MRPT*-a, prethodna vrednost se dodaje vrednosti sa cilem da se generiše prediktovana memorijska adresa. Prediktovana adresa se zatim puni u *PQ*-u. Ulazi u *PQ*-u se izvlače radi spekulativnog pristupa kešu podataka i ako se javi keš promašaj, tada se pristupa glavnoj memoriji ili narednom nivou keša. Prtistup kešu podataka je u suštini keš operacija, tj. pristup kešu se ostvaruje po redosledu kako bi se trigerovao potencijalni keš promašaj , a ne sa ciljem da se ostvari izvlačenje podataka iz keša.



Slika 4.49 *Prefetching* keš podataka

Napomena:: *Prefetch queue*- red čekanja tipa *prefetch* (pre-pribavljanja instrukcija)

Cilj *prefetching* keša ogleda se u njegovom pokušaju da učestvuje u nailazeće keš promašaje i da ranije trigeruje ove promašaje tako da se ublaži cena koja se plaća zbog keš promašaja preklapanjem ponovnih punjenja keša sa procesiranjem instrukcije koje prethode instrukciji *Load* zbog koje se javio promašaj. Kada se izvrši *Load* zbog koje se javio promašaj, podatak biće rezidentan u kešu, pa stoga će doći do generisanja keš promašaja, pa zbog toga cena koja će postojati zbog keš promašaja biće prihvatljiva. Aktuelna efikasnost *prefetching* zavisi od većeg broja faktora. *Prefetching* razmak, tj. koliko unapred se *prefetching* trigeruje mora biti dovoljno veliki da bi u potpunosti maskirao cenu koja se plaća zbog promašaja. Ovo je razlog da se pribavljanje adrese prediktovane instrukcije koristi za pristup *MRPT*-u, sa nadom da će podatak koji se pribavlja biti dovoljno ispred instrukcije *Load*. Na žalost ovo čini da *prefetching* efikasnost bude podložna efikasnosti predikcije grananja. Šta više postoji opasnost od zagadjivanja keša podataka sa unapred pribavljenjim instrukcijama koje se nalaze na pogrešno prediktovanom putu. Statusni ili bitovi poverljivosti se mogu dodati svakom ulazu *MRPT*-a kako bi modulisale agresivnost *prefetching*-a. Drugi problem koji se javlja kada se *prefetching* obavlja mnogo ranije sastoji se u iseljenju korisnog bloka iz keša i u izbegavanju nepotrebnog promašaja. Dodatni faktor predstavlja korišćenje aktuelnog algoritma za predikciju obraćanja. Predikcija *Load*

adrese zasnovana na napredovanje je veoma efikasno za *Load* instrukcije koje napreduju jedna za drugom kroz polje. Za ostale *Load* koje prolaze kroz strukture podataka tipa lančane liste predikcija neće raditi tako dobro. *Prefetching* kod ovakvih obraćanja memoriji zahteva znatno sofisticiranije algoritme za predikciju.

Da bi se poboljššao memorijski protok podataka, instrukcije *Load* se moraju izvršiti što je moguće pre. *Store* instrukcije su manje važne jer na osnovu eksperimentalnih podataka se došlo do zaključka da se redje javljaju u odnosu na *Load* instrukcije i da se one ne nalaze na kritičnom putu sa aspekta performansi. Da bi ubrzali izvršenje instrukcije *Load* mi moramo smanjiti latenciju procesiranja *Load* instrukcija. Ukupna latencija kod procesiranja *Load* instrukcija sadrži sledeće četiri komponente:

1. protočna *front-end* latencija za *fetching, decoding,* i *dispatching Load* instrukcije,
2. latencija rezervacione stanice zbog čekanja razrešavanja registarske *zavisnosti po podacima*,
3. latencija *execution* protočnog sistema kod generisanja i transliranja adrese, i
4. latencija kod pristupa kešu radi izvlačenja podataka iz memorije.

Oba tipa keša, *nonblocking* i *prefetching* imaju uticaj samo na četvrtu komponentu, koja je ključna komponenta zbog veoma spore memorije. Da bi radili sa većom taktnom frekvencijom, superskalarni protočni procesori postaju sve dublji i dublji. Saglasno tome latencije, u zavisnosti od broja mašinskih ciklusa, prvih triju komponenti takodje postaju značajne. Veći broj spekulativnih tehnika je predložen sa ciljem da se smanje ove latencije, tu spadaju predikcije adrese *Load*, predikcija *Load* vrednosti, i predikcija memorijske zavisnosti.

Od skoro predložen je veći broj tehnika za predikciju *Load* adrese (*load adrress prediction*) koje imaju za cilj da razreše latencije koje se odnose na prve tri komponente. Da bi izašli na kraj sa latencijom koja prati prvu komponentu predlaže se korišćenje *load prediction table* koja je slična *MRPT*-u. Ova tabela se indeksira sa prediktovanom pribaljenom adresom instrukcije, a pogodak u ovoj tabeli označava prisustvo instrukcije *Load* u dolazećoj *fetch* grupi. Stoga predikcija prisustva *Load* instrukcije u dolazećoj *fetch* grupi se obavlja u stepenu *fetch* bez da se zahtevaju usluge stepena za dekodiranje i dispečovanje. Svaki ulaz u ovu tabelu sadrži prediktovanu efektivnu adresu koja je izvučena iz stepena *Fetch*, čime se eliminiše potreba za čekanje u rezervacionoj stanici na dostupnost vrednosti baznog registra i stepena za generisanje adresa u *execution* delu protočnog sistema. Saglasno tome, pristup kešu podatka može početi u narednom ciklusu, a potencijalni podatak se može izvlačiti iz keša na kraju stepena *Decode*. Ovaj oblik *Load* adresne predikcije može da efikasno da smanji latencije prvih triju komponenti na dva ciklusa, tj. *Fetch* i *Decode* stepeni, pod uslovom da je predikcija adrese korektna i da se javio pogodak u toku obraćanja kešu podataka.

I pored toga što hardverske strukture za podršku predikcije *Load* adresa imaju veoma slične strukture onima koje se koriste za *prefetching* instrukcija iz memorije, izmedju ova dva mehanizna postoje značajne razlike. Dok sa jedne strane predikcija *Load* adrese u suštini spekulativno izvršava *Load* instrukciju ranije, sa druge strane *prefetching* instrukcije iz memorije pokušava da pribavi unapred neophodne podatke u keš bez da se izvrši *Load* instrukcija. Kod predikcije *Load* adrese, instrukcije koje zavise od *Load* se mogu izvršiti ranije pošto su njihovi zavisni podaci ranije i dostupni. Imajući u vidu da je predikcija *Load* adrese spekulativna tehnika, ona se mora validirati, pogrešna predikcija detektovati, i nakon toga oporavljanje obavljati. Validacija se obavlja na taj način što se omogućava da se stvarna *Load*instrukcija pribavi iz intrukcionog keša i izvrši na normalan način. Rezultat spekulativne verzije se komparira sa onim koji je karakterističan za normalnu verziju. Ako dodje do slagana rezultata, tada spekulativni rezultat postaje ne spekulativni i sve zavisne instrukcije koje su psekulativno izvršene deklarišu se kao ne spekulativne. Ako dodje do neslaganja rezultata, tada se koristi ne spekulativni rezultat, a sve zavisne instrukcije se moraju ponovo izvršiti. Ako je mehanizam za predikciju *Loadi* adresa zadovoljavajuće tačan tada se pogrešne predikcije retko javljaju, cena koja mora da se plati zbog loše predikcije je mala a povećanje performansi značajno.

Nešto agresivniji pristup u odnosu na predikciju *Load* adrese predstavlja predikcija *Load* vrednosti. Nasuprot predikciji *Load* adrese koja pokučšava da predvidi efektivnu adresu *Load*

instrukcije predikcija *Load* vrednosti pokušava da predvidi vrednost koja se izvlači iz memorije. Ovo se ostvaruje proširenjem *Load* predikcione tabele tako da ona sada sadrži ne samo prediktovanu adresu nego i prediktovanu vrednost odredišnog registra. Eksperimentalni rezultati su pokazali da je kod velikog broja *Load* instrukcija odredišna vrednost veoma predvidljiva. Na primer, veliki broj *Load* instrukcija puni istu vrednost kao zadnjeg puta. Zbog toga memorisanje zadnje vrednosti koja se puni od strane statičke *Load* instrukcije u *load prediction table* ukazuje da se ova vrednost može koristiti kao prediktovana vrednost kada se *Load* instrukcija ponovo izvršava. Kao rezultat, *load prediction table*-i se može pristupati u toku stepena *Fetch*, pa na kraju tog ciklusa, aktuelna odredišna vrednost prediktovane *Load* instrukcije može postati dostupna za korišćenje u narednom ciklusu za potrebe zavisne instrukcije. Na ovaj način značajno se smanjuje potrebna latentnost za procesiranje *Load* instrukcije pod uslovom da je prediktovana *Load* vrednost korektna. I sada ponovo, neophodna je validacija, a za slučaj pogrešne predikcije mora se platiti odredjenja cena.

Pored tehnike tipa predikcije *Load* adrese i tehnike predikcija *Load* vrednosti, veoma često se koristi i spekulativna tehnika nazvana predikcija memorijske zavisnosti (*memory dependence prediction*). Napomenimo, na osnovu izlaganja u sekciji 4.3.4, da bi obavili *load bypassing* i *load forwarding* potrebno je da se proveri memorijska zavisnost. Kod *load bypassing* mora se odrediti da *Load* nije u alijazi sa bilo kojom od *Store* koje su *bypassed* (premošćene). Kod *load forwarding* najskorija alijaza zbog *Store* instrukcije mora biti identifikovana. Provera memorijske zavisnosti zbog toga može biti veoma kompleksna pogotovu ako veći broj *Load/Store* instrukcija je uključen u proveru i ako potencijalno zahteva ugradnju jednog celog protočnog stepena. Poželjno je svakako eliminisati ovu latenciju. Eksperimenti pokazuju da su memorijske zavisnosti veoma predvidljive. Moguće je pratiti memorijske zavisnosti koje postoje kada se *Load/Store* instrukcije izvršavaju i koristiti ovu informaciju da bi se obavila predikcija memorije pod uslovom da se isti programski redosled *Load/Store* instrukcija ponavlja. Predikcija ovakvih memorijskih zavisnosti može da olakša ranije izvršenje *load bypassing* i *load forwarding*. Kao i kod spekulativnih tehnika, neophodno je ugraditi mehanizam za validaciju kao i mehanizam za oporavljanje od pogrešne predikcije.