Novica Nosović i Željko Jurić

OSNOVE RAČUNARSKIH ARHITEKTURA

Naslov

Osnove računarskih arhitektura

Za izdavača

Prof.dr.sc. Narcis Behlilović

Recenzenti

Prof.dr.sc. Slavko Marić, Univerzitet u Banja Luci, Elektrotehnički fakultet

Doc.dr.sc. Saša Mrdović, Univerzitet u Sarajevu, Elektrotehnički fakultet.

DTP i izrada

Autori

Štampa

Elektronska forma

Tiraž

Elektronska forma

Odlukom Senata Univerziteta u Sarajevu br.:01-38-1479-3/12 od 06. 06. 2012. godine, dana je saglasnost da se knjiga OSNOVE RAČUNARSKIH ARHITEKTURA izda kao univerzitetsko izdanje.

CIP - Katalogizacija u publikaciji
Nacionalna i univerzitetska biblioteka
Bosne i Hercegovine, Sarajevo

004.2(075.8)

NOSOVIĆ, Novica
Osnove računarskih arhitektura [Elektronski izvor] / Novica Nosović, Željko Jurić. - Elektronski tekstualni podaci. - Sarajevo : Elektrotehnički fakultet, 2012. - 1 elektronski optički disk (CD-ROM) : tekst, slike, animacije ; 12 cm

Tekst s nasl. ekrana.

ISBN 978-9958-629-50-1
1. Jurić, Željko
COBISS.BH-ID 19647750

Univerzitetski udžbenik
"Osnove računarskih arhitektura"
autora
Novice Nosovića & Željka Jurića

Novica Nosović
Željko Jurić

Osnove računarskih arhitektura

Sarajevo, 2012.

The important thing is not to stop questioning.

ALBERT EINSTEIN

Predgovor

Obzirom na veoma intenzivan razvoj računarske tehnike, naročito u toku protekle decenije, pisanje udžbenika koji će na zadovoljavajući način obraditi tematiku vezanu za računarske arhitekture predstavlja izazov, ali u isto vrijeme predstavlja i hrabru odluku. Ovaj udžbenik, pod naslovom “Osnove računarskih arhitektura”, predstavlja pionirski poduhvat čiji je cilj da se po prvi put na području Bosne i Hercegovine ova materija prezentira na akademski prihvatljiv način, pogodan za izvođenje nastave na visokoškolskim institucijama. Udžbenik predstavlja zaokruženu cjelinu koja je nastala na osnovu radnih materijala i iskustva kojeg su autori stekli tokom višegodišnjeg rada u nastavi na raznim predmetima iz oblasti računarske tehnike odnosno digitalne tehnike općenito na Elektrotehničkom i Prirodno-matematičkom fakultetu Univerziteta u Sarajevu.

Ovaj udžbenik prvenstveno je namijenjen studentima Elektrotehničkog fakulteta Univerziteta u Sarajevu, ali ga mogu koristiti i svi drugi koji posjeduju elementarno predznanje iz oblasti računarske tehnike i koje zanima problematika koja se u njemu razmatra. Udžbenik u potpunosti prati nastavni plan i program predmeta “Računarske arhitekture” koji studenti Odsjeka za računarstvo i informatiku na Elektrotehničkom fakultetu u Sarajevu slušaju kao obavezan kurs u IV semstru I ciklusa studija, a isti kurs se može odabrati i kao izborni kurs na višim godinama studija Odsjeka za matematiku Prirodno-matematičkog fakulteta u Sarajevu, smjer teorijska kompjuterska nauka. Pored ovog kursa, za koji ovaj udžbenik predstavlja obaveznu literaturu, udžbenik također može biti od koristi kao dopunska literatura na drugim kursevima koji se bave problematikom računarske tehnike. Ovo se prvenstveno odnosi na kurseve “Digitalni računari – arhitektura i organizacija softvera” i “Projektovanje mikroprocesorskih sistema” koji kao redovne kurseve slušaju studenti smjera za automatiku i elektroniku na Elektrotehničkom fakultetu u Sarajevu, a studenti drugih smjerova ih mogu odabrati kao izborne kurseve. Također, neka poglavlja udžbenika mogu olakšati razumijevanje nekih tema koje se obrađuju na kursevima “Operativni sistemi” i “Sistemsko programiranje” koji se slušaju na Odsjeku za računarstvo i informatiku na Elektrotehničkom fakultetu u Sarajevu, kao i na Odsjeku za matematiku Prirodno-matematičkog fakulteta u Sarajevu, smjer teorijska kompjuterska nauka (prvi kao obavezni, a drugi kao izborni kurs).

Materija obrađena u udžbeniku organizirana je u 12 poglavlja. Prvo poglavlje predstavlja uvodno poglavlje koje uvodi čitaoca u problematiku računarskih arhitektura. Nakon upoznavanja sa osnovnim pojmovima vezanim za računarske arhitekture, slijedi prikaz hardverskih i softverskih apstrakcija, položaja arhitekture skupa instrukcija među drugim apstrakcijama, te prikaz komunikacije između procesora i memorije. Poglavlje završava kratkim osvrtom na tematiku poglavlja koja slijede.

Drugo poglavlje posvećeno je istorijskom razvoju računara. Poglavlje započinje opisom karakteristika pojedinih generacija računara. Zatim se razmatraju tehnološki trendovi te i istorijskim datumima i akterima događaja koji su obilježili razvoj računarstva. U nastavku slijedi prikaz nekoliko različitih načina podjele računarskih arhitektura. Posebno se razmatra Von Neumannova arhitektura, kao vjerovatno najviše proučavana arhitektura svih vremena, nakon čega se opisuje kako je dalje tekla evolucija razvoja računarskih arhitektura od Von Neumannove arhitekture ka savremenim arhitekturama. Na kraju poglavlja, izlažu se osnove dizajna računara. Tom prilikom dati su i neki načini mjerenja performansi računarskih sistema.

U trećem poglavlju razmatra se arhitektura skupa instrukcija. Kroz ovo poglavlje se daje uvod u probleme koji se javljaju prilikom odlučivanja koje i kakve instrukcije procesor treba da izvršava, te koje formate podataka i načine adresiranja treba da podrži. Također se

objašnjava i uloga koju kompajleri imaju na savremene arhitekture skupa instrukcija.
Nakon apstraktnih razmatranja provedenih kroz prva tri poglavlja, u četvrtom poglavlju se opisuje jedan konkretni ogledni procesor zasnovan na arhitekturi sa protočnom strukturom. Prvo se detaljno opisuje instrukcijski set i format instrukcija, nakon čega se razmatra efektivnost prikazane ogledne arhitekture. Posebna pažnja posvećena je razmatranju protočnosti. Detaljno je prikazana podjela posla u protočnoj strukturi i ubrzanja koja se time postižu, kao i formati instrukcija koje takva organizacija diktira. Izvršeno je poređenje sa sličnom arhitekturom bez protočne strukture.
Peto poglavlje posvećeno je hazardima, koji predstavljalju osnovne probleme koji se javljaju u protočnim strukturama. Tom prilikom data je podjela hazarda sa detaljnim opisom svake od pojedinih vrsta hazarda. Također su prikazani i neki načini za rješavanje razmotrenih problema. Detaljno je opisan mehanizam upravljanja protočnom strukturom kao i problemi koje donose prekidi, izuzeci, te višeciklusne operacije nad brojevima sa pomičnim zarezom.
U šestom poglavlju razmatra se paralelizam na nivou instrukcija. Posebna pažnja se posvećuje
otkrivanju paralelizama na nivou instrukcija i petlji u kodu, kao i načina njihovog iskorištavanja, sa ciljem postizanja viših performansi računarskog sistema. Za razliku od statičkog raspoređivanja instrukcija, u ovom poglavlju se uvodi i razmatra dinamičko raspoređivanje instrukcija. Tom prilikom razmatraju se dva osnovna načina dinamičkog raspoređivanja instrukcija, sa semaforom i pomoću Tomasulovog algoritma.
Sedmo poglavlje opisuje dinamičko predviđanje grananja, kao pokušaja rješavanja usporenja koje grananja i skokovi unose u izvršenje koda. Posebna pažnja daje se korištenju bafera odredišnih adresa grananja kao strategiji za smanjenje kašnjenja kod instrukcija grananja.
Osmo poglavlje razmatra moguću podršku kompajlera u povećanju paralelizma na nivou instrukcija. Ovo poglavlje nastavlja se na teme iz prethodnog poglavlja i analizira ograničenja koja sprečavalju dalje povećanje i iskorištenje postojećih paralelizama. Posebno su prikazane
statičke (kompajlerske) i dinamičke (hardverske) metode ubrzavanja izvršenja programa kao što su softverske protočne strukture, trasiranje, kompajlersko spekulisanje uz podršku hardvera, hardversko spekulisanje, itd.
U devetom poglavlju razmatraju se superskalarne arhitekture procesora kao i aritekture procesora sa veoma dugim instrukcijskim riječima (VLIW procesori). U pitanju su pokušaji da se što više posla pokrene i izvrši u svakom ciklusu sata. Posebno se razmatra višestruko pokretanje instrukcija sa dinamičkim raspoređivanjem, kao i ograničenja procesora koji koriste višestruko pokretanje.
Deseto poglavlje je posvećeno memorijskoj hijerarhiji. U ovom poglavlju se detaljno opisuju mehanizmi keširanja kao i principi virtuelne memorije. Također se analizira i interakcija ova dva podsistema.
Podsistem ulazno-izlaznih uređaja razmatra se u jedanaestom poglavlju. Tom prilikom se razmatraju performanse ulazno-izlaznih uređaja i njihova uska grla. Detaljno je opisan princip rada magnetnih diskova, kao i nekoliko vrsta sabirnica koji se koriste za povezivanje ulazno-izlaznih uređaja sa centralnom procesnom jedinicom. Na kraju poglavlja razmatraju se mehanizmi koji olakšavaju upravljanje ulazno-izlaznim uređajima. Posebna pažnja data je sistemu prekida i mehanizmu direktnog pristupa memoriji (DMA pristupu).
Posljednje, dvanaesto poglavlje daje uvod u paralelne računarske arhitekture kroz dva osnovna principa – sisteme sa dijeljenom i distribuiranom memorijom. Tom prilikom prikazuju se principi projektovanja paralelnih računara, zatim načini povezivanja parova
procesor-memorija i drugih komponenti sistema, te mreže za povezivanje i preklopnici. Na kraju je dat osvrt na ograničenja u performansama takvih sistema.

Za razliku od većine udžbenika prisutnih na našem tržištu, ovaj udžbenik je dostupan *isključivo u elektronskoj formi*, koji po potrebi korisnici mogu štampati u vlastitoj režiji. Ovakav pristup predstavlja priličnu novinu na ovim prostorima. Pored činjenice da se na taj način olakšava dostupnost udžbenika svim zainteresiranim, dodatna prednost takvog pristupa je što se eventualno uočene sitnije greške i propusti u materijalu mogu "u hodu" dotjerivati, tako da će korisnici uvijek moći preuzeti najnoviju verziju. Naravno, ovdje se govori samo o izmjenama koje ne remete postojeći koncept udžbenika, jer u slučaju promjene koncepta, neće biti govora o novoj verziji, već o novom izdanju udžbenika.

Autori se duboko zahvaljuju recenzentima dr. Slavku Mariću i dr. Saši Mrdoviću što su se prihvatili zahtjevnog i odgovornog posla recenzije ovog udžbenika i na korisnim primjedbama. Pored toga, autori se zahvaljuju i svim drugim ljudima koji su pomogli izdavanje ovog udžbenika.

Autori

U Sarajevu, 18. 06. 2012. godine

1. Uvod u računarske arhitekture

Danas se računari koriste u svim sferama života. Postali su sastavni dio većine ljudskih dnevnih aktivnosti. Mogu se naći u različitim oblicima i veličinama. Tipični današnji računar ima više procesorske snage, memorijskog prostora i bolju komunikaciju sa ulazno-izlaznim uređajima nego tipičan računar iz 1950-ih, kada su se računari pojavili na tržištu, iako su tada zauzimali prostor jedne veće učionice. Postoje primjene za koje današnji tipični računar nije dovoljno moćan. Primjera za to ima više: prognoza vremena, simulacije nuklearnih procesa, proračuni u astronomiji i bioinformatici. Tu je potrebna mnogo veća računarska snaga. S druge strane, postoje mnogo manji računari koje i ne primjetimo u svakodnevnom životu. Skriveni (ugrađeni – eng. *embedded*) su u kućanskim aparatima, automobilima, mobilnim telefonima, projektorima koji se koriste na predavanjima, pa čak i u njihovim daljinskim upravljačima.

Postavlja se pitanje šta je zajedničko svim tim vrstama računara koje srećemo u ovom vremenu i prostoru, a šta je različito? Koji su zajednički principi na kojima počivaju svi današnji računari, kao i oni iz prošlih vremena?

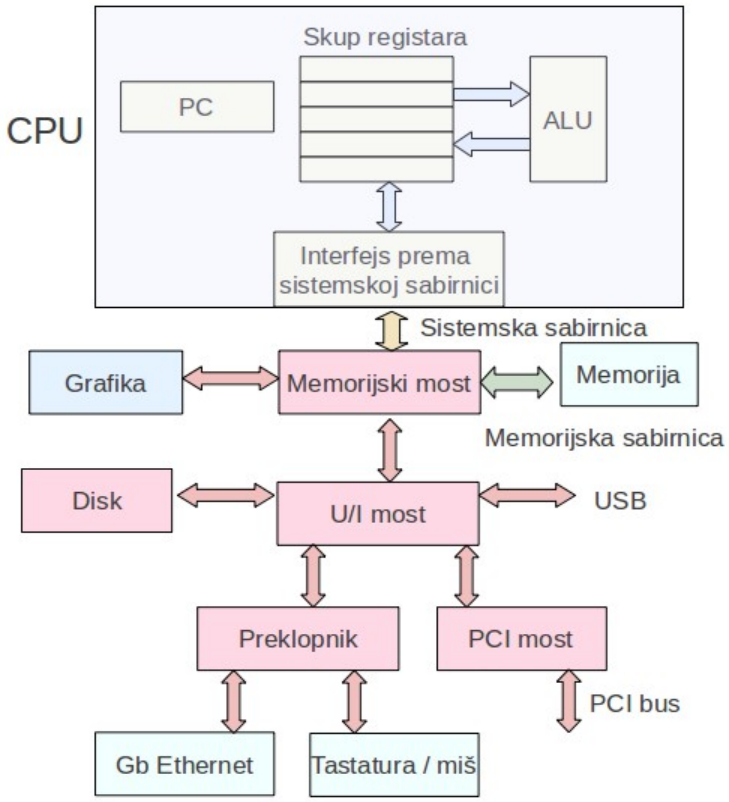
Odgovore na ta i neka druga pitanja daje oblast računarskih nauka koju nazivamo računarske arhitekture (od latinskog „architectura“, od starogrčke složenice „arhitekton“, od ἀρχι *glavni* i Τεκτον *građa*). Ona objašnjava kako rade računari i koji su osnovni principi na kojima počivaju sve vrste računara. Bavi se i razjašnjavanjem efikasnosti rada računara i njegovim performansama, o čemu ljudi često imaju pogrešna mišljenja. Postoje i drugi važni faktori, kao što je potrošnja energije, veličina (cijena) čipova i neki drugi o kojima treba voditi računa kada se govori o arhitekturama računara i njihovim performansama.

Pojam računarske arhitekture su još početkom 1960-tih godina različiti autori definisali na različite načine. U to doba glavni projektant u IBM-u (za IBM/360 sistem) G.M. Amdahl je koristio taj pojam da “opiše osobine sistema viđene od strane programera, tj. konceptualnu strukturu i funkcionalno ponašanje, za razliku od organizacije protoka podataka i upravljačke strukture, logičkog dizajna i fizičke implementacije”. H. S. Stone je rekao:“S*tudij računarskih arhitektura je studij organizacije i međuveza komponenti računarskih sistema*”. Ova definicija bi se mogla upotpuniti kada bi se posmatrala u smislu načinâ na koji se hardver računara može organizovati u cilju postizanja maksimalnih performansi, mjereno npr. prosječnim vremenom izvršenja instrukcija. Na taj način arhitekta računarskog sistema, ispoljavajući svoju “vještinu oblikovanja” glavne strukture, traži kompromisna rješenja - tehnike kojima će, umjerenom cijenom i složenošću hardvera, postići povećanje ukupnih performansi sistema.

Nakon razumjevanja osnovnih pojmova o tome kako rade računari, potrebno je znati i kako ih projektovati i napraviti. Koje složene tehnike se koriste tom prilikom i čemu koja služi? Naprimjer, kako keš memorija utiče na ostatak memorijske hijerarhije ili kako protočne strukture izvršavaju instrukcije paralelno.

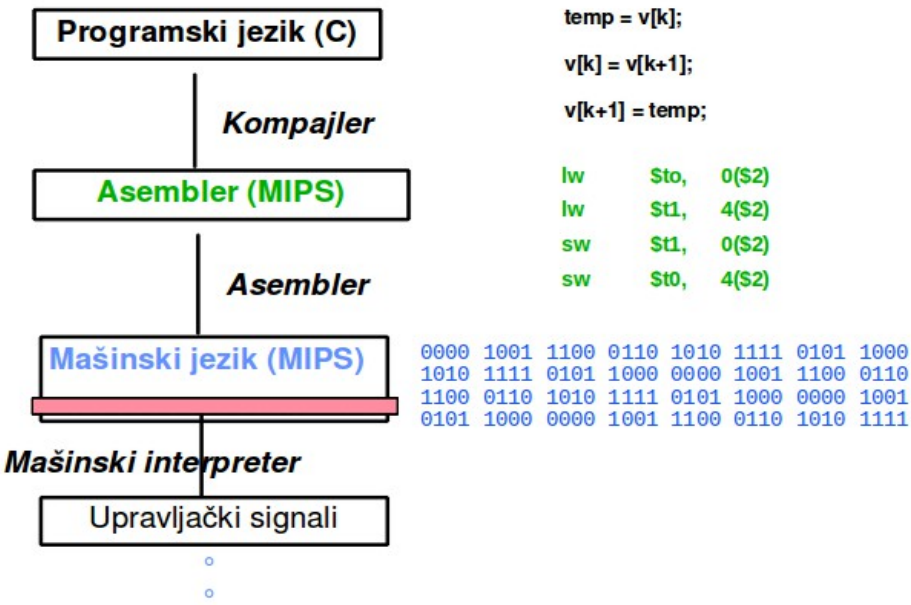
Zašto sve ovo učiti? Hoće li neko od nas imati priliku razvijati novi procesor ili računar? Vjerovatno ne, ali je vjerovatno da ćete doći u situaciju da projektujete neko unapređenje ili novu verziju postojećeg sistema. S druge strane razvoj softvera će vam biti mnogo vjerovatnije sastavni dio radnog vijeka. Tada će se postavljati zahtjevi za efikasno izvršenje ili postizanje maksimalnih performansi softvera. Za to su potrebna znanja o računarskim arhitekturama i hardveru računara, kako bi računarski sistem (hardversko-softverska cjelina) efikasno radila. S druge strane, za rukovodioce računarskih firmi i administratore računarskih

sistema, pomenuta znanja su neophodna. Razumjevanje principa rada sistema pomaže pri njegovoj nabavci, korištenju, održavanju i zanavljanju.
Postoji druga interesantna oblast, ugrađeni računari, gdje su računari dio sistema, u kojoj se otvaraju mnoge mogućnosti za projektovanje i razvoj. Razvoj novih aplikacija za takve računare, kao komponente sistema, predstavlja inžinjerski izazov.
Pojam računarskih arhitektura je najlakše objasniti povezivanjem sa građevinskim arhitekturama. U građevinarstvu arhitektura zgrade je plan rasporeda prostorija i njihove upotrebe. Za stambenu kuću to su naprimjer: dnevna soba, spavaća, kuhinja, kupatilo itd. Time je dat plan funkcionalnosti različitih dijelova zgrade. Nakon toga, građevinski inženjer će na osnovu toga plana uraditi svoj dio posla kroz strukturni dizajn zgrade, proračune nosivosti, stabilnosti, trajnosti itd. Ova analogija važi i za odnos arhitekte računara i inžinjera koji projektuje logičke sklopove - komponente računara. Prema tome arhitektura je plan ukupne funkcionalnosti. U računarskim arhitekturama to znači koje osnovne operacije će se izvoditi i kako će se one sekvencirati. Za realizaciju ovakvog plana/arhitekture je potreban dizajner računarskih sklopova koji će objediniti sve komponente (tranzistore, otpornike i druge) u jednu funkcionalnu cjelinu. Zato arhitekta računara mora imati znanja i iz projektovanja logičkih struktura, iako projektuje na višem nivou apstrakcije.
Apstrakcije se javljaju u raznim oblastima života, inžinjeringa i nauke. Razlika između spoljašnjih (korisničkih) osobina neke komponente i njenih unutrašnjih (dizajnerskih) detalja zove se **apstrakcija**. Ona je važna tehnika pojednostavljivanja bez koje bi život u civilizovanom svijetu bio nezamisliv. Civilizovani čovjek ne mora znati kako su realizovane razne pogodnosti savremenog svijeta. On jede hranu i oblači odjeću koju ne zna kako napraviti. On koristi električne i elektronske uređaje čiju unutrašnjost ne poznaje. Koristi tuđe usluge čiji detalji aktivnosti mu nisu poznati. Sa svakim novim napretkom, mali dio društva se specijalizuje da bi spoznao njegovu unutrašnjost (napraviti i održavati), dok veliki dio društva uči koristiti taj napredak kao apstraktni alat-sretstvo čiju unutrašnju građu ne mora poznavati. Sa povećanjem broja takvih apstraktnih alata, šansa društva da napreduje - raste.
U računarskim athitekturama postoji više nivoa-slojeva apstrakcije. Apstrakcija znači izostavljanje manje važnih detalja u cilju boljeg razumjevanja najvažnijih stvari i problema. Ulazeći u dubinu apstrakcija količina podataka se povećava, dok sa više apstrakcija količina informacija se reducira. Između arhitekture i sklopova postoji više slojeva apstrakcije, kao što postoji i između softvera i hardvera. Apstrakcije pomažu u savladavanju složenosti. Pravljenje računara od najosnovnijih komponenti je gotovo neizvodljivo. Softver pisan u jeziku visokog nivoa (apstrakcije!) se pomoću kompajlera prevodi na niži nivo apstrakcije - asemblerski kod, gdje se sekvenciraju mnemonici instrukcija. Na još nižem nivou apstrakcije se taj kod asemblerom prevodi u mašinski kod, sekvencu nula i jedinica, koju računar razumije. Hardver u sebi sadrži više nivoa apstrakcije (slika 1.1.)



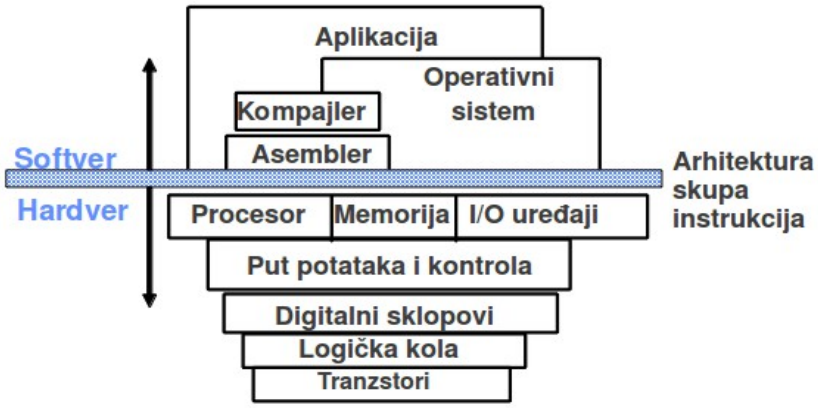
Slika 1.1. Apstrakcije u hardveru (blok struktura PC-a)

Glavni moduli su CPU (procesor), memorija, ulazno-izlazni uređaji i njihovi kontroleri. Svi su vezani u sistem preko sabirnica i kontrolera. Detaljniji ulaz u procesor otkriva registre, programski brojač (PC), aritmetičko-logičku jedinicu (ALU) i druge komponente, kao i unutrašnje veze među njima. Dalji ulaz u svaku od tih komponenti otkrivao bi složene logičke sklopove koji se sastoje od osnovnih logičkih komponenti, a one od tranzistora. Tranzistori se mogu razložiti na svoje komponente i tako se spušta na još niži nivo apstrakcije. Slika 1.2. predstavlja hijerarhiju apstrakcija između softvera i hardvera.



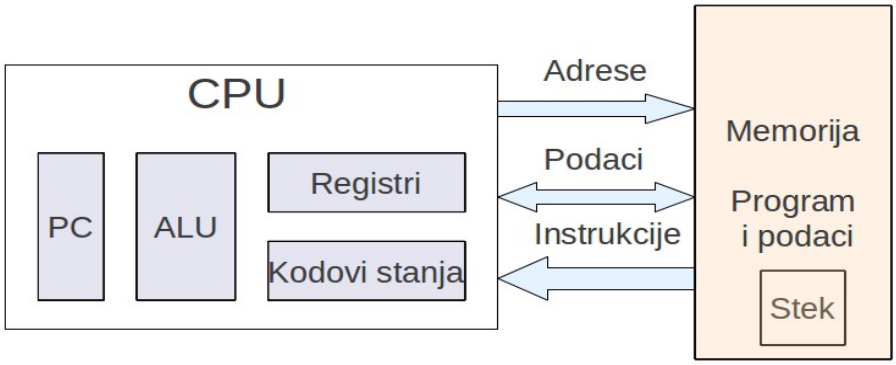
Slika 1.2. Hardver/softver interfejs

Na najvišem nivou je program pisan u jeziku visokog nivoa. Na najnižem nivou softvera su mašinske instrukcije procesoru. Između njih je asemblerski sloj. U hardverskom dijelu najviši nivo su glavni sastavni blokovi- registri, sabirači (ALU) sabirnice, dok su na dnu osnovne komponente - tranzistori. Fokus računarskih arhitektura je u okolini tanke linije koja predstavlja granicu između hardvera i softvera. Tu se nalazi skup instrukcija koji određuje osnovne operacije procesora, sa jedne strane i glavne komponente procesora koje razumiju te instrukcije, sa druge strane. Programer vidi procesor kroz skup instrukcija, dok dizajner hardvera vidi softver kroz sekvencu mašinskih instrukcija koje treba interpretirati (izvršiti). Sa obje strane navedene tanke linije se nalazi više slojeva apstrakcija - kako u softveru, tako i u hardveru. Arhitektura računara je definisana interfejsom između softvera i hardvera i, kao nauka, najčešće se bavi najnižim slojem softverske i najvišim slojem hardverske hijerarhije apstrakcija - arhitekturom skupa instrukcija (ISA) i mikroarhitekturom hardvera procesora. ISA se odnosi na najniži nivo apstrakcije vidljiv programeru. Programer ne treba da brine o tranzistorima, flip-flopovima i logičkim kolima, već samo o osnovnim jedinicama računanja - instrukcijama. Mikroarhitekturom se bavi dizajner hardvera i njen zadatak je da se brine o logičkom povezivanju instrukcija i hardverskih modula. Pojedini moduli (skup registara, ALU) sami za sebe ne definišu skup instrukcija, već način na koji su oni međusobno povezani u jedinstvenu cjelinu - put podataka. Položaj ISA među hardverskim i softverskim nivoima apstrakcije je prikazan na slici 1.3.



Slika 1.3. Položaj arhitekture skupa instrukcija među hardverskim i softverskim apstrakcijama

Na vrhu su aplikativni programi koji se izvršavaju na procesoru pomoću sistemskog softvera. Kompajler prevodi program iz jezika visokog nivoa u mašinski kod, dok operativni sistem upravlja resursima i omogućava rad kompajliranje, punjenje i izvršavanje programa. Ispod ISA nivoa se nalazi dizajn centralne procesne jedinice (CPU), a na nižem nivou logički dizajn sklopova. Na dnu je fizički dizajn sa razmještajem komponenti na silicijumu.
Pogled na procesor kroz asemblerski jezik jednoznačno određuje u kojem stanju je procesor. Njegovo stanje u svakom trenutku je određeno stanjem registara i sadržajem memorije. Nakon izvršenja svake instrukcije procesor prelazi iz jednog u drugo stanje. U memoriji se nalaze instrukcije i podaci programa koji se izvršava. Kopije instrukcija koje se trenutno izvršavaju, kao i podataka nad kojima se te operacije izvode se nalaze u procesoru (registrima). Nakon završetka obrade, rezultati se pišu u memoriju.
Iznad asemblerskog nivoa apstrakcije nalaze se kompajler i operativni sistem, kao i jezici visokog nivoa. Ispod njega se nalaze hardverske komponente visokog nivoa (skup registara, ALU itd) i nižeg nivoa (tranzistori, otpornici) koje omogućavaju izvršenje svih instrukcija u mašinskom jeziku. Arhitekte imaju zadatak da organizuju i povežu ove hardverske komponente tako da se instrukcije izvršavaju bez greške i što je moguće brže. Pored performansi, sve češće treba voditi računa i o drugim ograničenjima, kao što je potrošnja energije, što je naročito važno kod prenosnih uređaja koji se napajaju iz baterija.
Osnovna blok-struktura i princip rada računara je data na slici 1.4.



Slika 1.4. Par procesor-memorija i njihova osnovna komunikacija

U memoriji se nalazi kod u mašinskom jeziku i podaci koje treba obraditi. Procesor mora biti u stanju donijeti instrukcije i podatke iz memorije i nakon obrade vratiti rezultate natrag. Obično se u memoriji nalazi i posebna struktura – stek, koji služi za pozivanje funkcija, procedura i, generalno, pomaže u kreiranju hijerarhije programskih apstrakcija. Osnovne komponente CPU-a su programski brojač (PC), ALU, skup registara kao i registar sa kodom stanja (eng. CC – Condition Code). ALU radi aritmetičko-logičke operacije nad operandima iz registara, pa operande prethodno treba donijeti iz memorije. PC pokazuje na instrukciju koju sljedeću treba donijeti iz memorije i izvršiti. Nakon toga pokazuje na sljedeću i na taj način obezbjeđuje sekvenciranje instrukcija. Instrukcije se međusobno razlikuju po tome kako su kodirane, koje operacije izvršavaju i šta u njima ALU treba da radi. U CPU-u postoji i upravljačka jedinica, koja nije prikazana na slici 1.4., koja upravlja svim pomenutim komponentama, kako bi odradile odgovarajuće poslove u odgovarajuće vrijeme. Izvršenje instrukcija ažurira i stanje CC registra, koji je najčešće dio skupa registara, čije stanje se kasnije može koristiti za donošenje odluka u programu kao što su grananja i petlje poznate iz jezika visokog nivoa.

Pomenute komponente su zajedničke za sve procesore, jednostavne i složene. Postoje arhitekture procesora koje donekle odstupaju od ovakve organizacije, ali se može reći da je ovo najčešća organizacija procesora posljednjih decenija.

Apstrakcija ovakvog procesora koji ima relativno jednostavnu strukturu i izvršava jednostavne operacije pomaže u razumjevanju osnovnih koncepata. Međutim, kao kod svih apstrakcija, vrlo je važno razumjeti šta se apstrahuje u nekom trenutku i koja je razlika između date apstrakcije i realnih struktura u svakodnevnom životu. Ignorisanje nekih detalja može dovesti do velikih grešaka u razumjevanju čitave tehnologije. Primjera za to je mnogo. U prirodi je broj cjelobrojnih vrijednosti neograničen. U računarima (hardveru i programima), za njihovo predstavljanje se koristi ograničen broj bita pa je i njihov opseg ograničen. Sličan problem postoji i u predstavljanju realnih brojeva, gdje ograničenja u broju bita mantise i eksponenta direktno nameću ograničenja u rezoluciji i opsegu vrijednosti koje se mogu predstaviti. Stoga je sasvim moguće da zakoni matematike prestaju važiti ukoliko se računanja sa realnim brojevima odvijaju na digitalnom računaru. Primjer za to može biti sabiranje tri realna broja od kojih neki predstavljaju vrlo velike a neki vrlo male vrijednosti. Od redosljeda sabiranja može zavisiti tačnost rezultata, što se kosi sa zakonom asocijativnosti. Greška može biti vrlo mala, ali mogućnost njenog nastanka postoji. Da bi se dobro poznavao rad jednog procesora, neophodno je poznavati njegov asemblerski jezik iako se u njemu danas vrlo rijetko programira. Danas se gotovo svi programi pišu u jezicima visokog nivoa, pa ipak, dobar programer koji želi da razumije ponašanje i performanse programa koji piše, mora razumjeti i šta se dešava u procesoru (na nižem nivou apstrakcije). U protivnom, programer je odvojen od fizičke realnosti izvršenja programa.

Apstrakcija memorije podrazumjeva neograničen broj riječi i bajta/bita u njima, sa istim vremenom pristupa svakoj riječi. U realnosti je sasvim drugačije. Ne samo da su riječi ograničene širine, a memorija ograničenog broja riječi - kapaciteta, već se pokazuje da je, u jednom trenutku, za pristup nekoj lokaciji potrebno znatno manje vremena nego nekoj drugoj, dok se nešto kasnije može desiti i obrnuto. U realnosti, memorija ima višenivosku hijerarhijsku strukturu, pa vrijeme pristupa različitim lokacijama može značajno varirati. i o tome dobar programer mora voditi računa. Vrijeme izvršenja nekog programa je parametar koji se želi maksimalno reducirati, kako bi se performanse arhitekture na kojoj se on izvršava maksimalno iskoristile.

U oblasti ugrađenog računarstva, dizajn sistema u kome je računar samo jedan dio (često nevidljiv) podrazumjeva, najčešće, projektovanje i hardvera i softvera – kodizajn. To je

mnogo teže od rada na samo jednoj od ove dvije komponente. U takvom okruženju, procesor se može posmatrati kao inteligentna (programabilna) elektronska komponenta, a ne kako mašina za "mljevenje" brojeva. Zato njihova procesorska snaga nije u prvom planu, već su osobine kao što je mala potrošnja energije i lakoća programiranja naročito cijenjene. Od takvih računara se očekuje rad u realnom vremenu – obrada podataka čim se oni pojave. Veliki broj ugrađenih računara pruža mogućnost njihovog prilagođavanja novim potrebama, bilo da se radi o doradi procesora, perifernih uređaja ili softvera koji se na njemu izvršava. Ovo podrazumjeva znanje dizajnera u oblasti i hardvera i softvera i to na nižim nivoima apstrakcije.
Na visokom nivou apstrakcije, procesor je isti za sve tipove računara. U stvarnosti, postoje značanje razlike među procesorima koji se koriste u serverima, stonim računarima, laptopima, mobilnim telefonima, mašinama za pranje suđa ili veša i u drugim uređajima. Iako svi rade na istom principu, imaju svoj skup instrukcija i izvršavaju ih jednu po jednu. Ono po čemu se razlikuju jeste, brzina izvršenja instrukcija, potrošnja električne energije, cijena, a skup instrukcija im je ili prilagođen opštim ili specijalnim primjenama.

1.1. Sadržaji poglavlja koja slijede

Poglavlje 2. govori o istoji računara, tehnološkim trendovima, kao i istorijskim datumima i akterima događaja koji su obilježili razvoj računarstva. Navedeno je i nekoliko načina podjele računarskih arhitektura, kao i neki načini mjerenja performansi računarskih sistema.
Poglavlje 3. daje uvod u probleme koji se javljaju prilikom odlučivanja koje i kakve instrukcije procesor treba da izvršava, koje formate podataka i načine adresiranja treba da podrži. Objašnjena je i uloga koju kompajleri imaju na savremene arhitekture skupa instrukcija.
Poglavlje 4. opisuje jednu oglednu arhitekturu procesora sa protočnom strukturom. Detaljno je prikazana podjela posla u protočnoj strukturi i ubrzanja koja se time postižu, kao i formati instrukcija koje takva organizacija diktira.
Poglavlje 5. predstavlja osnovne probleme koji se javljaju u protočnim strukturama – hazadre. Data je podjela hazarda, kao i neki načini njihovog rješavanja. Opisan je mehanizam upravljanja protočnom strukturom i problemi koje donose prekidi, izuzeci, kao i višeciklusne operacije nad brojevima sa pomičnim zarezom.
Poglavlje 6. je posvećeno otkrivanju paralelizama na nivou instrukcija i petlji u kodu, kao i načina njihovog iskorištavanja, sa ciljem postizanja viših performansi računarskog sistema. Preikazana su dva osnovna načina dinamičkog raspoređivanja instrukcija, sa semaforom i pomoću Tomasulovog algoritma.
Poglavlje 7. opisuje dinamičko predviđanje grananja, kao pokušaja rješavanja usporenja koje grananja i skokovi unose u izvršenje koda.
Poglavlje 8. je nastavak teme iz prethodnog poglavlja i analizira ograničenja koja sprječavalju dalje povećanje i iskorištenje postojećih paralelizama. Posebno su prikazane statičke (kompajlerske) i dinamičke (hardverske) metode ubrzavanja izvršenja programa.
Poglavlje 9. daje osnovne pojmove iz oblasti superskalarnih arhitektura i arhitektura sa vrlo dugom instrukcijskom riječi (VLIW), kao pokušajima da se što više posla pokrene i izvrši u svakom ciklusu sata.
Poglavlje 10. je posvećeno memorijskoj hijerarhiji. Posebno su opisani mehanizmi keširanja kako i principi virtuelne memorije. Analizirana je i interakcija ova dva podsistema.
Poglavlje 11. analizira podsistem ulazno-izlaznih uređaja, njihove performanse i uska grla. Dat je opis principa rada magnetnih diskova, nekoliko vrsta sabirnica i mehanizama koji

olakšavaju upravljanje ulazno-izlaznim uređajima – sistem prekida i direktnog pristupa memoriji.

Poglavlje 12. daje uvod u paralelne računarske arhitekture kroz dva osnovna principa – sisteme sa dijeljenom i distribuiranom memorijom. Prikazani su principi povezivanja parova procesor-memorija, kao i ograničenja u performansama takvih sistema.

2. Istorija računara

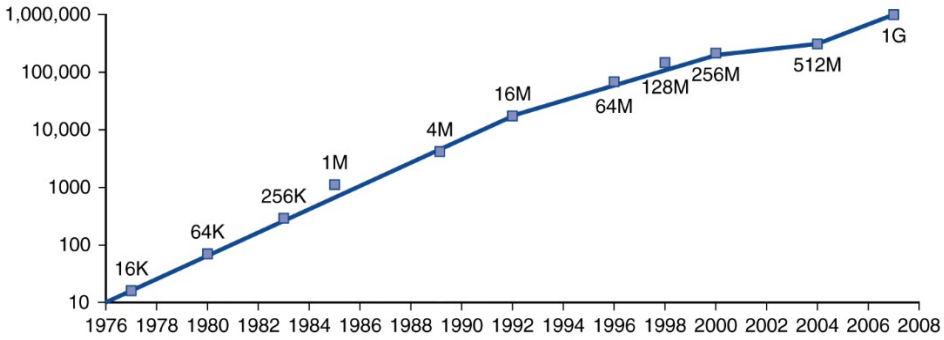
Istorija računara je važna da bi se razumjela hronologija razvoja tehnologija koje su uticale na razvoj računarskih arhitektura. Na osnovu tih znanja lakše je razumjeti sadašnjost i budućnost računarskih komponenti i sistema.
U proteklih 50 do 70 godina računari se mogu podijeliti na 4 do 5 generacija. Svaku je obilježio neki značajan tehnološki napredak. Bilo da se radilo o napredku u veličini, cijeni, potrošnji energije, efikasnosti ili pouzdanosti, on je omogućio novi iskorak u istoriji računara.
Napredak u veličini je omogućio da za smještaj računara više nije potrebna veća učionica, već da se on može smjestiti na sto, u krilo ili čak na dlan.
Napredak u cijeni je u velikoj mjeri povećao pristupačnost i omogućio da se od situacije kada su računare mogli priuštiti samo vojska i velike državne institucije, dođe do toga da danas ima više računara nego ljudi na planeti (računajući i ugrađene).
Potrošnja energije u računarima se drastično smanjila, za nekoliko redova veličine, i time povećala efikasnost, performanse i pouzdanost računarskih sistema. Od računara iz 1950-tih godina, koji su imali srednje vrijeme između otkaza reda 20-tak minuta, pouzdanost se toliko popravila da današnji (mnogo složeniji) računari, naročito serveri, rade godinama bez greške.
Prva generacija računara, iz 1940-ih i 1950-ih godina, je bila bazirana na elektronskim vakuumskim cijevima kao prekidačkim elementima. Računari su bili skupi, glomazni, nepouzdani i trošili su enormne količine električne energije. Kao ulazno-izlazne uređaje koristili su bušene kartice ili trake, a memorija je realizovana na rotirajućim magnetnim bubnjevima. Programirani su u mašinskom jeziku pomoću spojnih kablova i višepoložajnih prekidača.
Druga generacija računara, iz 1950-ih i 1960-ih godina (uz preklapanje u vremenu sa prethodnom), je bila bazirana na tranzistorima kao prekidačkim elementima. Oni su bili mnogo manji, brži, jeftiniji, energetski efikasniji i pouzdaniji od vakuumskih cijevi. Programi za njih su pisani u asemblerskom jeziku, a pojavile su se i prve verzije jezika visokog nivoa – FORTRAN i COBOL.
Treća generacija računara, iz 1960-ih i 1970-tih godina, je bila bazirana na integrisanim kolima. U početku to su bila kola malog stepena integracije (SSI, od eng. Small Scale Integration, nekoliko desetina tranzistora na silicijumskom supstratu - čipu), dok se kasnije prešlo na srednji (MSI, od eng. Medium Scale Integration, nekoliko stotina tranzistora na čipu) i visoki stepen integracije (LSI, od eng. Large Scale Integration, nekoliko hiljada tranzistora na čipu). Sa povećanjem stepena integracije, računari su postajali manji, brži i efikasniji. Pojavom tastatura i monitora umjesto ranijih bušenih kartica, traka ili teleprintera, računari su postali pogodniji za interakciju sa ljudima. Bušenje kartica, slaganje i čekanje satima ili danima na obradu, otklanjanje grešaka i na kraju dobijanje rezultata je svedeno na nekoliko sekundi ili minuta. Tada su se pojavili i prvi ozbiljniji operativni sistemi koji su omogućavali rad više programa i više korisnika istovremeno i podigli efikasnost korištenja naraslih računarskih resursa.
Četvrta generacija računara, od 1970-ih do danas, je bazirana na mikroprocesorima. Nju je omogućila pojava integrisanih kola vrlo visokog stepena integracije (VLSI, od eng. Very Large Scale Integration, od stotinu hiljada tranzistora na čipu, nadalje). Pod isti pojam spadaju i današnji procesori sa više od milijardu tranzistora, jer termin ultra-visokog stepena integracije (ULSI, od eng. Ultra Large Scale Integration) nije zaživio. Od prve polovine 1970-ih, postalo je moguće čitav procesor smjestiti na jadan čip, čime je postao jedna

komponenta računarskog sistema. U drugoj polovini 1970-tih, padom cijena procesora, pojavili su se kućni računari, ugrađeni računati i počela je era ličnih (personalnih) računara. Rast performansi procesora omogućio je pojavu grafičkog korisničkog okruženja (GUI, od eng. Graphical User Interface) i miša (dobio ime po obliku i kablu koji liči na rep). Na tržištu su se počeli pojavljivati i prenosni uređaji (HHD, od eng. Hand Held Devices).
Mnogi autori i današnje računare smatraju pripadnicima četvrte generacije jer su i dalje bazirani na mikropocesorskim tehnologijama. Drugi pak današnje i buduće računare smatraju petom generacijom, ne toliko kao posljedicu razvoja hardverskih tehnologija, već prvenstveno po sve većem korištenju vještačke inteligencije kako bi se računari učinili jednostavnijim za upotrebu i sličnijim ljudima po načinu interakcije sa okolinom. Promijenjen je pogled na računarstvo, a podrazumjevaju se gotovo neograničeni računarski resursi. Primjeri za to su pokušaji da se glasom i prirodnim (ljudskim) jezicima komunicira sa računarom (razgovara). Zbog potrebe za visokim performansama, često se u ovu generaciju svrstavaju i paralelne računarske arhitekture.
Brzi rast performansi komponenti računara ilustruje tabela 2.1. Ona prikazuje relativan odnos performansi i cijena četiri generacije tehnologije izrade prekidačkih elemenata u rasponu od oko 40 godina.

Godina	Tehnologija	Odnos performanse/cijena
1951	Vakuumske cijevi	1
1965	Tranzistori	35
1975	Integrisana kol	900
1995	VLSI	2.400.000
2005	ULSI	6.200.000.000

Tabela 2.1. Rast relativnog odnosa performansi prekidačkih elemenata u odnosu na cijenu

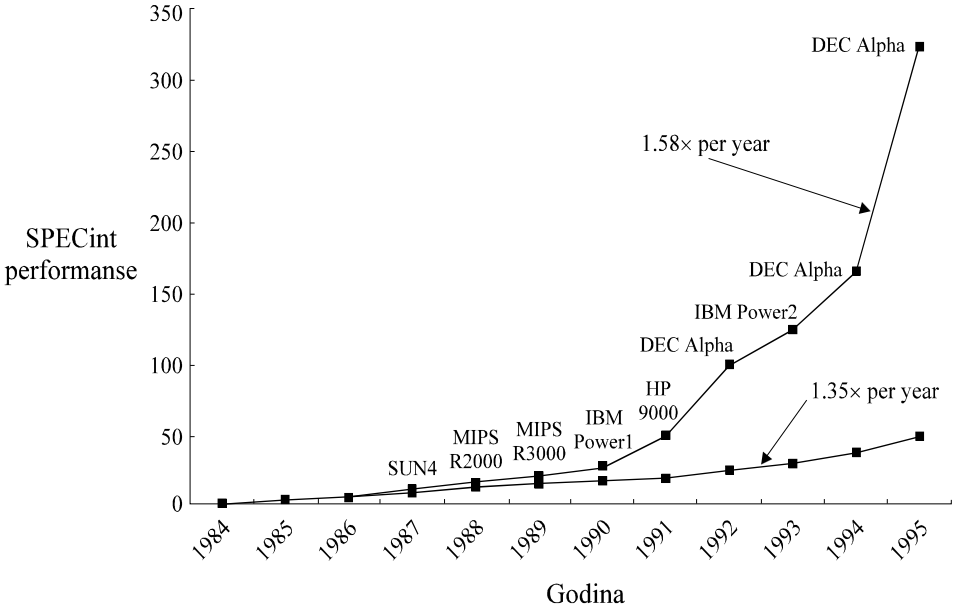
Napredak u tehnologiji koji je prikazan u ovoj tabeli je fascinantan i teško ga je naći u nekim drugim tehnologijama u istoriji.
Rast kapaciteta memorijskih komponenti je prikazan na slici 2.1.



Slika 2.1. Trend rasta kapaciteta memorijskih komponenti (broja tranzistora) u posljednje tri decenije

Vidi se konstantan rast kapaciteta, ali treba primjetiti da je on eksponencijalan (oznake na Y osi rastu eksponencijalno. Rast je sličan rastu frekvencije osnovnog signala sata kod

procesora. Udvostručava se svakih godinu do godinu i po.
Relativan rast performansi računarskih sistema je dat na slici 2.2. i pokazuje eksponencijalan rast, pogotovo od sredine 1980-tih godina, kada su arhitekture sa reduciranim skupom instrukcija (RISC, od eng. Reduced Instruction Set Computers) prevladale na tržištu.



Slika 2.2. Performanse mikroprocesora od sredine 1980-tih rastu znatno brže nego ranije. Do tada se rast oslanjao na tehnologiju izrade integrisanih struktura, a kasnije se pripisuje naprednim arhitekturama - naročito u aritmetici sa pomičnim zarezom.

Neki ključni događaji u razvoju računarskih sistema prikazan u tabeli 2.2.

Godina	Pronalasci i pronalazači	Opis
1936	Z1, Konrad Zuse	Prvi programabilni računar
1944	H. Aiken, Harvard Mark I	Harvard arhitektura
1946	Eckert/Mauchly, ENIAC 1	18000 Elektronskih cijevi
1948	Shockley, tranzistor	Velika promjena u istoriji računarstva
1951	Eckert/Mauchly, UNIVAC	Prvi komercijalni računar
1953	IBM 701	IBM ulazi u istoriju računarstva
1954	John Bacus/IBM, FORTRAN	Prvi uspješan jezik visokog nivoa
1958	J. Kilby, R. Noyce, integrisano kolo	Pojava “čipova”
1962	Steve Russel/MIT, Spacewar	Prva računarska igra
1964	Douglas Engelbart, miš i Windows	Napredni korisnički interfejsi
1969	ARPAnet	Preteča Interneta
1970	Intel 1103 memorija	Prvi DRAM čip
1971	Faggin/Hoff/Mazor, Intel 4004	Prvi mikroprocesor

1971	Alan Shugart/IBM, fleksibilni disk	Poznatiji kao “flopy”
1973	R. Metkalfe/Xerox, Ethernet mreža	Početak umrežavanja računara
1976	Apple I, II, Comodore	Početak masovne proizvodnje PC-a
1987	D. Bricklin/B. Frankston, VisiCalc	Tabelarni račun (spreadsheet) (isplatio se za dvije sedmice)
1979	S. Rubenstein/R. Barnaby, WordStar	Procesor teksta
1981	IBM PC – ozbiljan kućni računar	Početak revolucije personalnih računara
1981	Microsoft, MS-DOS	Početak revolucije operativnih sistema
1983	Apple Lisa	Prvi kućni računar sa grafičkim interfejsom
1984	Apple Macintosh	Pristupačan računar sa grafičkim interfejsom
1985	Microsoft Windows	Početak “rata” Microsoft - Apple

Tabela 2.2. Neki od važnih događaja u istoriji računarstva.

Da bi se razumjela perspektiva razvoja računarskih arhitektura, neophodno je razumjeti razvoj osnovnih komponenti računarskih sistema, a pogotovo njihove osnovne osobine kao što su propusnost i kašnjenje.
Projektanti (arhitekte) prvog velikog računarskog sistema MARK-1 (1943.) su bili ograničeni raspoloživom tehnologijom (Williams-ove cijevi kao memorije, problem pregrijavanja, a time i pouzdanosti), pa su pravili (logički) male i relativno jednostavne uređaje. Ipak je, složenim načinima adresiranja i mogućnošću hardverskog množenja (cjelobrojnog) “odskočio” od do tada uobičajenih dizajna. Nešto kasnije, 1957. god., je počeo da se komercijalno proizvodi model MERCURY sa ugrađenim hardverom za sabiranje i množenje brojeva u formatu sa pomičnim zarezom, zahvaljujući korištenju poluprovodničkih dioda, manjih ventila/prekidača nego kod MARK-1, kao i upotrebom feritnih jezgri za smještaj podataka. Krajem 1950-tih godina i početkom 60-tih, problem grijanja/pouzdanosti je u znatnoj mjeri otklonjen pojavom tranzistora na slobodnom tržištu. Tada su se pojavili prvi “superkompjuteri”: ATLAS (sa strananjem - virtuelnom memorijom), STRECH, MULTICS, pa sve do CDC 6600 (sa višestrukim funkcionalnim jedinicama) koji su ujedno istakli potrebu za složenijom programskom podrškom u obliku operativnih sistema, pogotovo za višekorisnička okruženja.

2.1. Performanse i podjele računarskih arhitektura

Povećati performanse računarskih arhitektura se može na tri osnovna načina:

1. tehnološkim unapređenjem,
2. efikasnijim korištenjem postojećih tehnologija i
3. unapređenjem komunikacije između hardvera i softvera.

Pod tehnološkim unapređenjem se podrazumjeva povećanje brzine logičkih kola i memorijskih struktura, kao i povećanje pouzdanosti. Upotreba tranzistora je omogućila da hardver u ATLAS-u bude brži od onoga u MERCURY-u jer su projektanti mogli priuštiti

paralelni sabirač za razliku od ranijeg mnogo sporijeg serijskog. Odgovarajućim napretkom u mogućim stepenima integracije postignuto je da se sve veća funkcionalnost može integrisati na mali prostor, pri čemu pouzdanost sistema ostaje manje-više nepromijenjena.

Efikasno korištenje postojećih tehnologija je glavni zadatak projektanata - arhitekata računarskog sistema, i postiže se, u glavnom, na sljedeće načine:

1. korištenjem višestruke hijerarhije memorije i
2. korištenjem istovremenosti/paralelizma (eng. concurency).

Hijerarhijskom organizacijom memorije nastoji se, uz pomoć hardversko-softverskih mehanizama dati "iluziju" korisniku (procesoru) da ima na raspolaganju veliku i brzu, a uz to i jeftinu memoriju.

Istovremenost/paralelizam se javlja u različitim formama i na različitim nivoima dizajna sistema. Na niskom nivou apstrakcije ona se ogleda kroz:

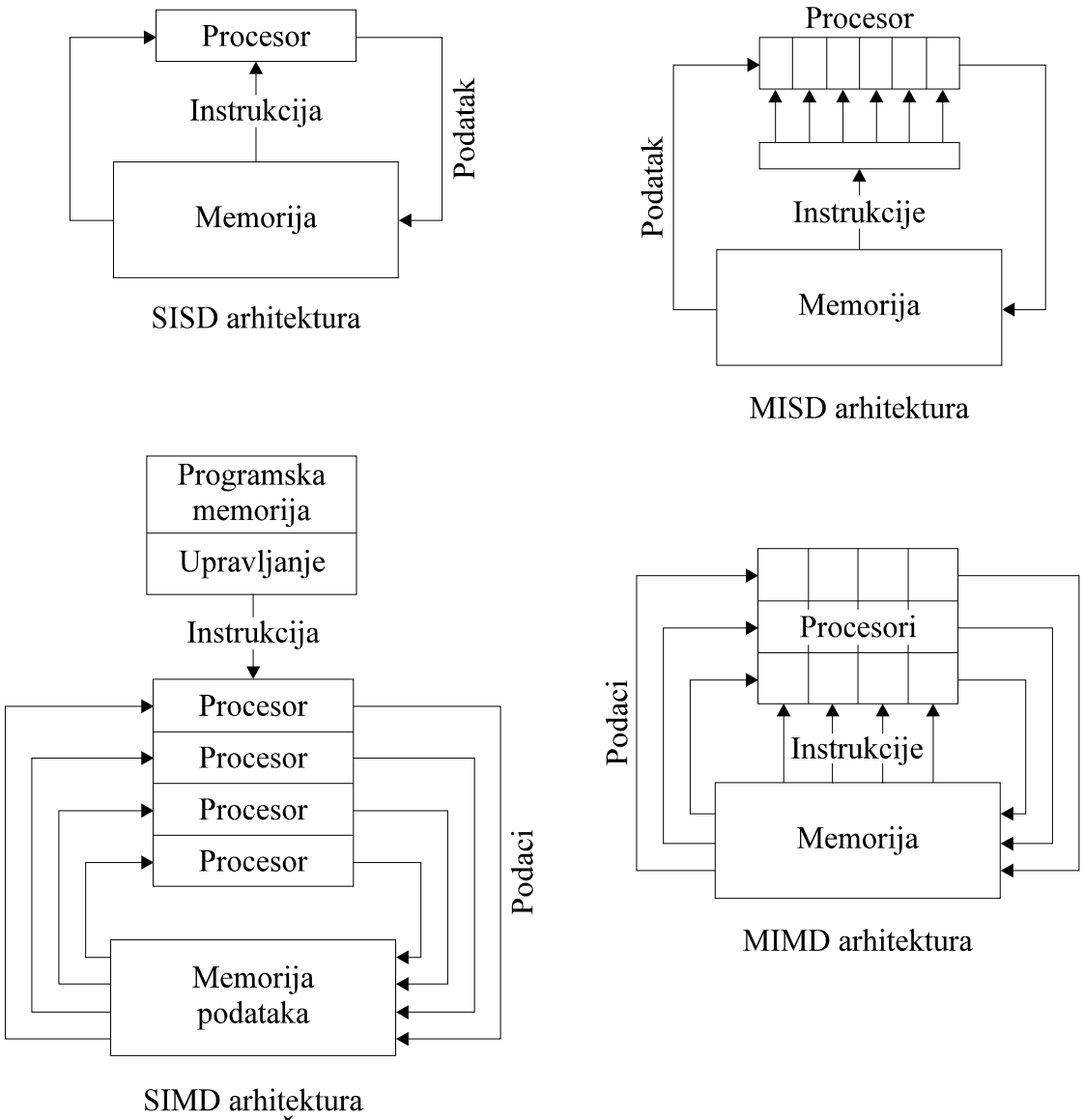
1. protočne tehnike (eng. pipeline),
2. preplitanje memorije i
3. paralelne funkcionalne jedinice

što je, sve skupa, nevidljivo programeru.

Na visokom nivou apstrakcije paralelizam obično podrazumjeva jedan broj procesora međusobno povezanih u niz ili mrežu, tako da djeluju "paralelno" pri rješavanju jednog zadatka.

Najpoznatija klasifikacija računarskih arhitektura na osnovu tipa paralelnosti koju ispoljavaju (slika 1.1), dao je Michael Flynn 1972. god. podjelom na:

1. **SISD** (eng. Single Instruction stream Single Data stream) - jednostruki instrukcijski tok, jednostruki tok podataka. Arhitektura SISD predstavlja arhitekturu sekvencijalnih računara von Neumannova modela računanja. Predstavnik tog tipa arhitekture su standardni serijski računari s jednim procesorom (uniprocessor), npr. DEC-ov računar PDP 11 ili računar CDC 3170.
2. **SIMD** (eng. Single Instruction stream Multiple Data stream) - jednostruki instrukcijski tok, višestruki tok podataka. Primjeri takvih računara su ILIAC IV, MPP (Massively Parallel Processor) i STARAN.
3. **MISD** (eng. Multiple Instruction stream Single Data stream) - višestruki instrukcijski tok, jednostruki tok podataka. Primjer ovog tipa arhitekture je protočni (eng. pipeline) računar, npr. IBM 360/91 i računar ASC tvrtke Texas Instruments.
4. **MIMD** (eng. Multiple Instruction stream Multiple Data stream) - višestruki instrukcijski tok, višestruki tok podataka. Računari ovog tipa arhitekture sastoje se od sistema više procesora u kojima procesori izvršavaju različite instrukcije nad različitim podacima. Primjeri takvih računara su multiprocesorski sistemi UNIVAC 1108 i C. mmp.



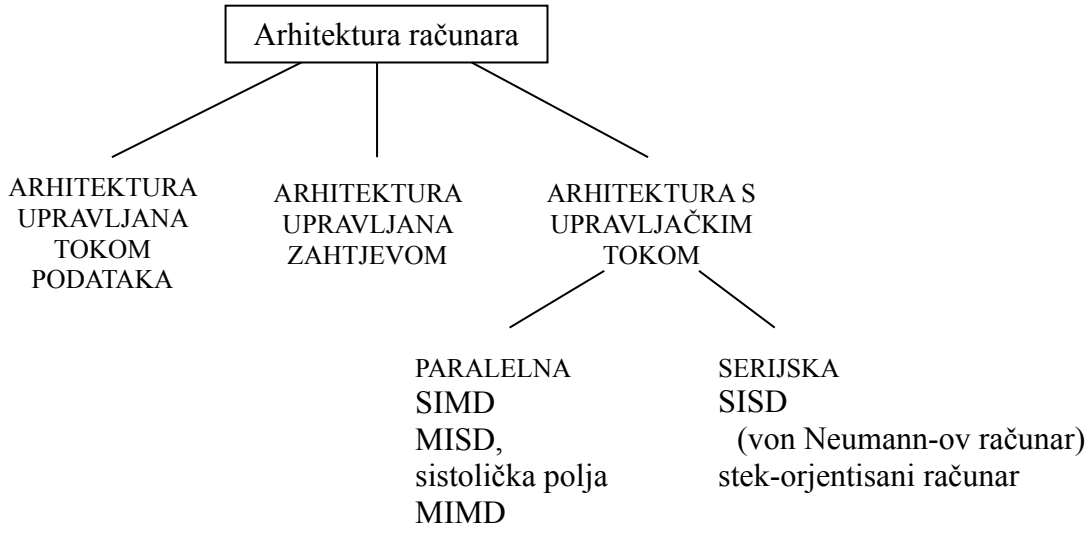
Slika 2.3 Šematski prikaz tipova arhitekture (prema Flynn-u).

Povećanje performansi računarske arhitekture unapređenjem komunikacije između hardvera i programske podrške se, u najvećoj mjeri, odnosi na pažljivi izbor skupa instrukcija (arhitekture skupa instrukcija). Kod nekih "moćnih" računara, "moć" se krije u hardveru i može se u punoj mjeri iskoristiti ili "ručnim" kodiranjem (u asembleru) ili korištenjem složenih algoritama u kompajlerima.

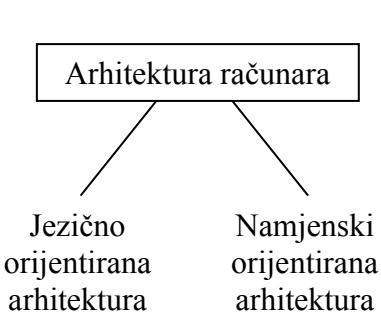
Arhitekture se mogu klasificirati na osnovu načina izvršavanja instrukcija na:

1 arhitekture sa upravljačkim tokom (eng. Control Flow)
2 arhitekture upravljane tokom podataka (eng. Data Flow) i
3 arhitekture upravljane zahtjevom (eng. Demand Driven).

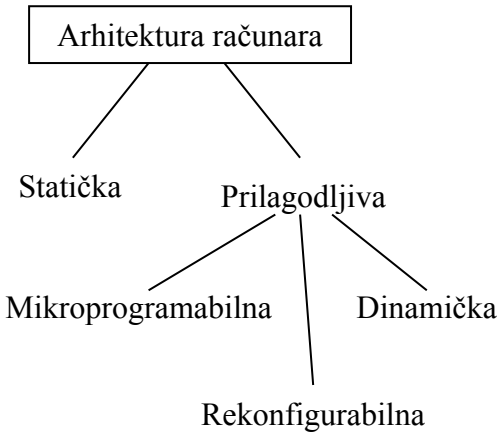
Slika 2.4. daje grafički prikaz nekoliko različitih klasifikacija računara.



a) Klasifikacija arhitekture računara s obzirom na:
način izvršavanja instrukcija
tok instrukcija i tok podataka



b) Klasifikacija arhitekture računara na osnovu odnosa viših programskih jezika i računara



c) Klasifikacija arhitekture računara na osnovu promjenljivosti strukture

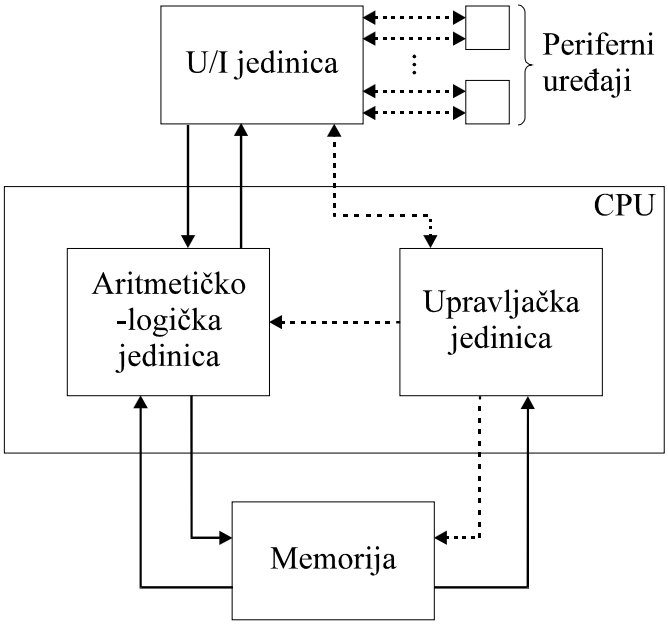
Slika 2.4. Različite klasifikacije arhitekture računara.

2.1.1. Von Neumann-ov model računara

Godine 1946., autori A. W. Burks, H. H. Goldstine i J. von Neumann, su objavili rad u kojem je detaljno opisan računar opšte namjene s programom smještenim u memoriju – zajedno sa podacima (eng. general purpose stored-program computer). Od tada se računarske arhitekture ovog tipa nazivaju Von Neumann-ovim modelom računara.

Slika 2.5. prikazuje model von Neumannova računara. On se sastoji od četiri funkcionalne jedinice:

1 aritmetičko-logičke jedinice (ALU),
2 upravljačke jedinice,
3 memorijske jedinice, i
4 ulazno-izlazne (U/I) jedinice.



Slika 2.5 Model von Neumann-ovog računara.

Aritmetičko-logička jedinica se sastoji od sklopova za izvršavanje osnovnih aritmetičkih operacija, i registara za privremeno pohranjivanje podataka (operanada) koji sudjeluju u operacijama.

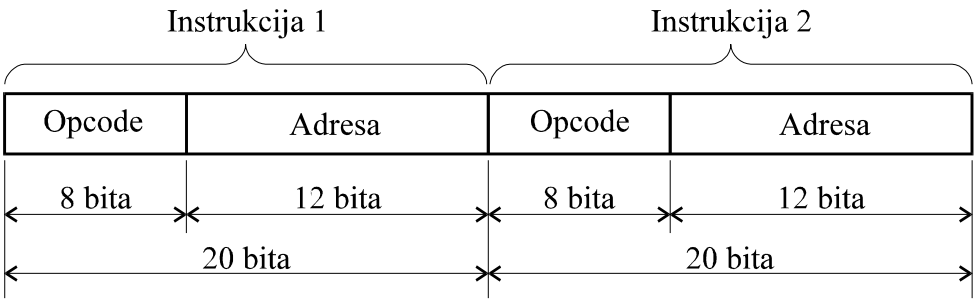
Za osnovu je uzet binarni brojni sistem (za razliku od do tada popularnog dekadnog brojnog sistema korištenog u računskim mašinama kao što je MARK-I).

Aritmetičko-logički sklopovi u prvom von Neumannovom računaru su bili sabirač i šifter. U radu s negativnim brojevima je korišten 2-komplement kod.

Operandi su imali dužinu 40 bita, što je omogućavalo preciznost računanja na dvanaest decimala (2^{-40} , tj. približno $0{,}9\times10^{-12}$), a kapacitet memorije je bio 4096 riječi.

Upravljačka jedinica je davala sve potrebne upravljačke signale za vremensko vođenje i upravljanje ostalim jedinicama računara. Program se izvršava tako da upravljačka jedinica pribavlja instrukcije u kodiranom obliku, dekodira ih i u skladu s njihovim značenjem generiše signale pomoću kojih ALU, memorija i U/I jedinica izvode potrebne operacije.

Von Neumann, Burks i Goldstine predložili su da se u memorijsku riječ dužine 40 bita smjeste dvije instrukcije (lijeva i desna instrukcija); svaka po 20 bita (slika 2.6.).



Slika 2.6. Organizacija instrukcijske riječi u von Neumannovom računaru.

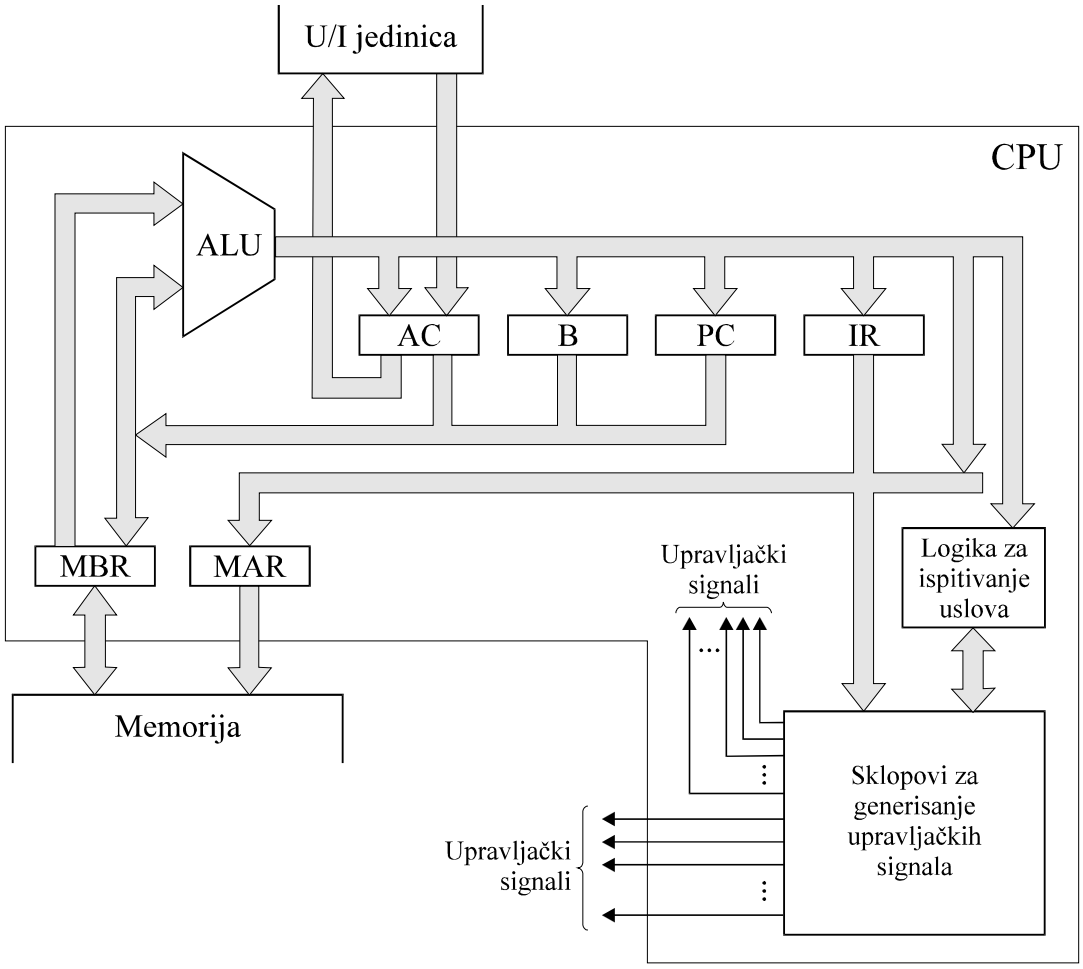
Prvih 8 bita instrukcije (Opcode) određuje operaciju koja će se izvršiti i dopušta 256 različitih kodova operacija, a sljedećih 12 bita je adresno polje koje određuje adresu operanda i omogućuje direktno adresiranje cijele memorije ($2^{12} = 4096$).

Instrukcijski skup je predviđao instrukcije za izvršavanje osnovnih aritmetičko-logičkih operacija, instrukcije za prenos podataka između ALU i memorije, instrukcije uslovnog i bezuslovnog grananja i instrukcije za rad s U/I uređajima. Tabela 2.3. daje pregled tipičnih instrukcija.

Instrukcija	Primjer	Operacija
Load	LOAD X	AC ← mem[X];
Store	STORE Y	mem[Y] ← AC;
Load negative	LNEG Z	AC ← –mem[Z];
Load absolute value	LABS W	AC ← ABS(mem[W]);
Add	ADD X	AC ← AC + mem[X];
Subtract	SUB Y	AC ← AC – mem[Y];
Multiply	MUL Z	AC ← AC × mem[Z];
Divide	DIV W	AC ← AC / mem[W];
Branch left	BRAL X	PC ← X.left_instruction;
Branch right	BRAR Y	PC ← Y.right_instruction;
Branch positive left	BPOSL X	if AC≥0 then PC ← X.left_instruction;
Branch positive right	BPOSR Y	if AC≥0 then PC ← Y.right_instruction;
Store address left	STADL X	mem[X].left_address ← AC;
Store address right	STADR Y	mem[Y].right_address ← AC;
Shift left	SHL	AC ← AC × 2;
Shift right	SHR	AC ← AC / 2;

Tabela 2.3. Pregled tipičnih instrukcija Von Neumann-ovog računara

Slika 2.7 prikazuje detaljniju organizaciju centralne procesne jedinice, koja se u nekoliko detalja razlikuje od originalne zamisli von Neumannova računara, ali djeluje na potpuno jednak način.



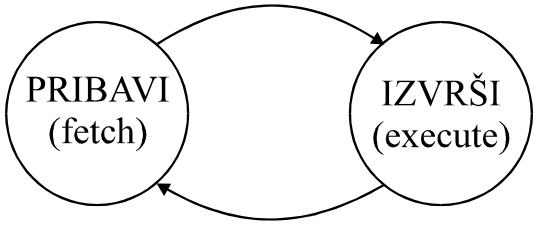
SLIKA 2.7. Organizacija centralne procesne jedinice von Neumannova računara.

Izvršavanje instrukcije se odvija u dvije faze: **PRIBAVI** (eng. fetch) i **IZVRŠI** (eng. execute).

U fazi PRIBAVI događa se sljedeće:
1. KORAK: Iz memorije se pribavlja (čita) sljedeća instrukcija i prenosi se u instrukcijski registar (IR). Adresa instrukcije se nalazi u registru PC.
2. KORAK: Sadržaj registra PC se povećava za jedan i određuje instrukciju koja neposredno slijedi za instrukcijom koja je upravo pročitana.
3. KORAK: Dekodira se 8-bitni kôd operacije. Pobuđena je samo ona izlazna linija dekodera kojoj odgovara kôd operacije pribavljene instrukcije.

S trećim korakom se završava faza PRIBAVI. Računar prelazi u fazu IZVRŠI:
4. KORAK: Pobuđuju se sljedovi operacija (npr. prenos podataka od memorije ili prema memoriji, prenos podataka prema registrima ili ALU, aktiviranje sklopova za izvršavanje aritmetičko-logičkih operacija, promjena vrijednosti PC u slučaju izvođenja instrukcije grananja) kojima se izvršava instrukcija.

Izvođenjem 4. koraka upravljačka jedinica se vraća na 1. korak, tj. u fazu PRIBAVI. Naizmjenični prelazak iz jedne faze u drugu se stalno ponavlja, sve dok se ne izvrši instrukcija za zaustavljanje rada (Halt). Slika 2.8. prikazuje dijagram prelaza stanja za von Neumann-ov računar.



SLIKA 2.8. Dijagram prelaza stanja za von Neumannov računar.

Von Neumannov model računara pretstavlja SISD arhitekturu, jer se u fazi PRIBAVI samo jedna instrukcija pribavlja iz memorije i ona se izvršava (faza IZVRŠI) nad podatkom, odnosno nad parom podataka koji se jedan iza drugog pribavljaju iz memorije.

2.1.2. Razvoj od Von Neumann-a

ALU - je evoluirala sa bit-serijske na bit-paralelnu obradu, dorađena je za rad sa FP-notacijom a povećanjem broja registara opšte namjene povećana je njena iskoristivost. Sa tehnološke strane prekidački sklopovi su ubrzani sa nekadašnjih reda sekundi na milisekunde pa mikrosekunde, nanosekunde pa sve do današnjih pikosekundi.

UPRAVLJAČKA JEDINICA - i njen dizajn su usko vezani sa izborom skupa instrukcija, mogućnostima grananja u programu, skokova u podprograme te željenim načinima adresiranja. U određenom periodu razvoja upravljačkih jedinica se javila potreba da se umjesto minimizacije broja logičkih kola ide na minimizaciju broja različitih logičkih tipova kola kako bi se postigla veća uniformnost (regularnost) strukture, a time veća gustina pakovanja, testabilnost, pouzdanost i još neke prednosti. Princip mikroprogramiranja (uveo M. Wilkes 1951. god.) je u velikoj mjeri olakšao projektovanje upravljačkih struktura i, između ostalog, omogućio jednostavniju podršku pri uvođenju sistema prekida (eng. interrupts).

MEMORIJA - od SELECTRON-a, katodnih cijevi za smještaj i čitanje podaka u originalnom Von Neumann-ovom dizajnu, pa do današnjih memorija, prošlo je više tehnoloških generacija: magnetne žice i bubnjevi, bušene kartice i trake, magnetne jezgre i bubble memorije. Kako je rastao kapacitet, brzina i pouzdanost memorijskih podsistema, projektanti su uvodili arhitekturalne novine kao što su virtuelne i CACHE memorije, strananje i preplitanje, što je dovelo do okruženja u kojima je bilo moguće realizovati multiprogramiranje - više nezavisnih programa se odvija istovremeno - koegzistira u memoriji.

ULAZNO/IZLAZNI UREĐAJI - su takođe prošli kroz nekoliko generacija. Od vremena kada je jedna upravljačka struktura upravljala istovremeno i procesorom i U/I uređajima do multiprocesorskih sistema kod kojih su procesori za upravljanje perifernim uređajima jednako moćni kao “glavni” procesor(i). Revolucionaran napredak je postignut uvođenjem vremenskog dijeljenja resursa (eng. time sharing) pri čemu svaki korisnik ima “iluziju” da je čitav računarski sistem njemu na raspolaganju.

2.2. Osnove dizajna računara

Današnji PC (1K$) ima bolje performanse, više memorijskog i diskovnog prostora nego mainframe-i iz 1960-tih i 1970-ih koji su koštali više miliona US$. Za tako brz rast je zaslužna:

1. tehnologija izrade komponenti (stalno raste) i
2. unapređenja u dizajnu/arhitekturi (nestabilno raste).

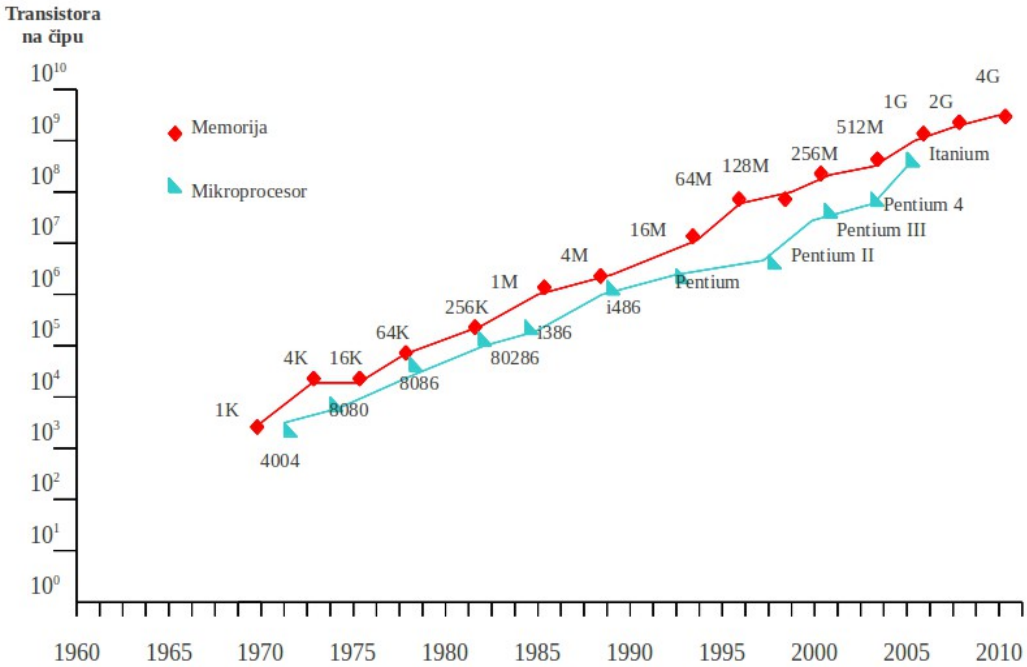
Od 1970-tih dizajneri sve više zavise od tehnologije izrade integrisanih kola. Do pojave mikroprocesora, performanse su rasle 25-30% godišnje, a od kraja '70-tih rast je 35%. Od tada gotovo svi dizajneri prelaze na mikroprocesore. Još dvije stvari utiču na brzi razvoj novih arhitektura:

1. gotovo niko više ne programira u asembleru, pa nema potrebe za kompatibilnošću na nivou source/objektnog koda,
2. novi operativni sistemi (UNIX i sl.) smanjuju rizik i cijenu uvođenja novih arhitektura.

Zato se, početkom 1980-tih pojavljuju RISC arhitekture. Od polovine '80-tih, nakon izlaska na tržište, njihove performanse rastu 50% godišnje (slika 1.8). To ima za posljedicu da:

1. korisnici dobijaju znatna poboljšanja performansi,
2. dominiraju mikroprocesorski bazirani dizajni (mainframe-e zamjenjuju multi-mikroprocesorski sistemi - čak i u području "superračunara").

Gordon Moore, jedan od osnivača Intel-a je još 1965. godine tvrdio da će se, u budućnosti, broj integrisanih tranzistora na čipu udvostručavati svake godine (kasnije je to promijenio u 18 mjeseci). Ilustracija ovog trenda je data na slici 2.9.



Slika 2.9. Moore-ov zakon povećanja broja tranzistora u odnosu na godine pojavljivanja pojedini verzija mikroprocesora i memorijskih komponenti.

Renesansa računarskih arhitektura je dovela do toga da je 1995. godine, odnos performansi tada vladajućih mikroprocesora i onoga što bi nastalo isključivo tehnološkim usavršavanjem klasičnih arhitektura veći od 1:5. Za nastavak ovakvog trenda je neophodan jasno definisan -

kvantitativni pristup projektovanju novih računarskih arhitektura.

Funkcionalni zahtjevi	Zahtijevane ili podržane karakteristike
Područje primjene	Cilj računara
Opšte namjene	Uravnotežene performanse za različite zadatke
Naučno	Visoke performanse u aritmetici sa pokretnim zarezom
Komercijalno	Podrška COBOL-u (decimalna aritmetika), podrška bazama podataka i procesiranju transakcija
Nivo softverske kompatibilnosti	Određuje količinu postojećeg softvera za mašinu
Na nivou programskog jezika	Najfleksibilnije za dizajnera, potreban novi kompajler
Objektna/binarna kompatibilnost	Arhitektura skupa instrukcija je potpuno definisana - malo fleksibilnosti - ne zahtijeva investicije u softver ili prevođenje programa
Zahtjevi operativnog sistema	Potrebne karakteristike za podršku izabranom OS-u
Veličina adresnog prostora	Vrlo važna osobina; može ograničiti primjenu
Upravljanje memorijom	Obavezno kod modernih OS-a - strananjem ili segmentiranjem
Zaštita	Različiti OS-i i aplikacije je zahtijevaju: stranična ili segmentirana
Standardi	Tržište može zahtijevati određene standarde
Aritmetika sa pokretnim zarezom	Format i arithmetika: IEEE, DEC, IBM
I/O sabirnice	Za U/I uređaje: VME, SCSI, Fiberchannel
Operativni sistemi	UNIX, DOS, ili neki specifični
Mreže	Zahtijevana podrška za različite mreže: Ethernet, ATM
Programski jezici	Jezici (ANSI C, Fortran 77, ANSI COBOL …) utiču na skup instrukcija

Tabela 2.4. Neki od najvažnijih zahtjeva s kojim se suočava arhitekta računarskog sistema. U lijevoj koloni su dati zahtjevi a u desnoj osobine arhitekture koje odgovaraju tim zahtjevima.

Zadatak dizajnera se svodi na dvije osnovne stvari:

1. odrediti koje su osobine nove mašine važne i
2. napraviti dizajn sa maksimalnim performansama u okviru date cijene.

Ranije je ovo podrazumjevalo:

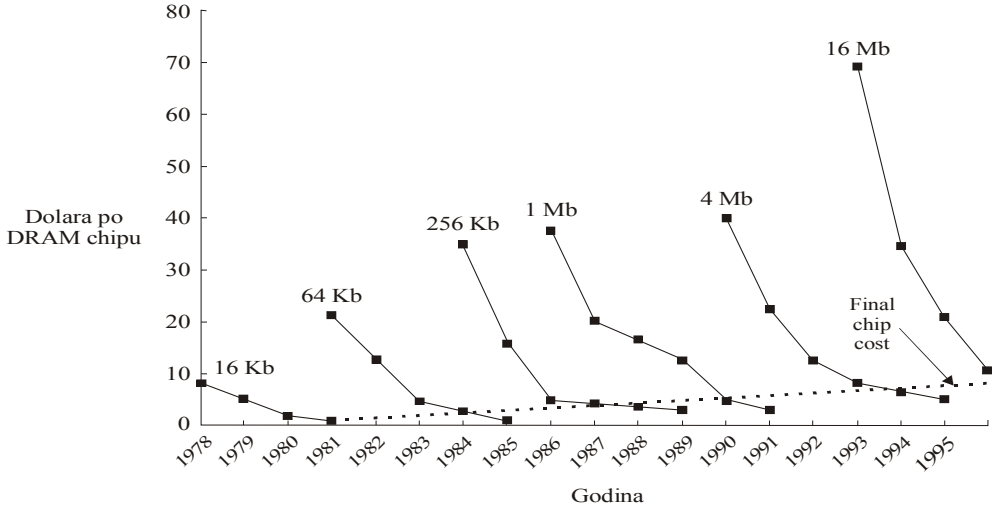
1. dizajn skupa instrukcija (ISA- Instruction Set Architecture), nešto manje,
2. funkcionalnu organizaciju,
3. logički dizajn i
4. implementaciju (IC-dizajn, pakovanje, potrošnja/disipacija-hlađenje i sl.)
5. pogotovo jer je razlika u ISA-ma među današnjim mikroprocesorima sve manja.

Kako odrediti koje su osobine nove mašine važne? To u najvećoj mjeri zavisi od:

1. njene namjene - područja primjene (opšta, naučna, komercijalna ili neka druga),
2. zahtjeva za programskom kompatibilnošću (na binarnom, objektnom ili na nivou jezika visokog nivoa (HLL - od eng. High Level Languages),
3. zahtjeva za podrškom OS-ima (memorijski prostor, virtuelna memorija-MMU, zaštita podataka ili nekih drugih),
4. zahtjeva za podrškom postojećim standardima (FPU, U/I, umrežavanje, programski jezici ili nekim drugim).

Trend korištenja računara pokazuje da potrebe za memorijskim prostorom rastu sa faktorom od 1.5 do 2 godišnje (1/2 do 1 adresni bit godišnje). Zbog lošeg planiranja ovog parametra propale su (finansijski) mnoge dobre računarske arhitekture. Prelaskom sa asemblera na programske jezike visokog nivoa važnost kompajlera naglo raste. Zato projektant i pisac kompajlera najčešće zajedno prave novu arhitekturu, ukoliko to ne radi jedna osoba. To se pogotovo odnosi na slučajeve izgradnje paralelnih arhitektura i odgovarajućih kompajlera.

Trend u tehnologiji izrade komponenti savremenih računarskih arhitektura pokazuje da gustoća pakovanja IC-a raste 50% godišnje, površina čipova 10-25%, što omogućuje da broj tranzistora raste 60-80%. Približno tome raste i brzina rada tih sklopova. Gustoća pakovanja DRAM-e raste 60% godišnje, dok je vrijeme pristupa smanjeno za 1/3 u zadnjih 10 godina. Za to vrijeme je njihova propusnost značajno porasla. Kapacitet diskova raste 50% godišnje, dok se vrijeme pristupa smanjuje brzinom kao kod DRAM-e. Ove tri tehnologije, uz sva dalja poboljšanja, daju dizajnu mikroprocesora životni vijek od oko 5 godina. Imajući na umu da dizajn obično traje 2 do 3 godine, da za relativno kratko vrijeme treba proizvesti i prodati dovoljno proizvoda da bi se na tome ostvario profit, a da na tržište treba izbacivati nove proizvode svakih 6 mjeseci do godinu dana, pokazuje se da je planiranje ovakvih projekata krajnje delikatno.

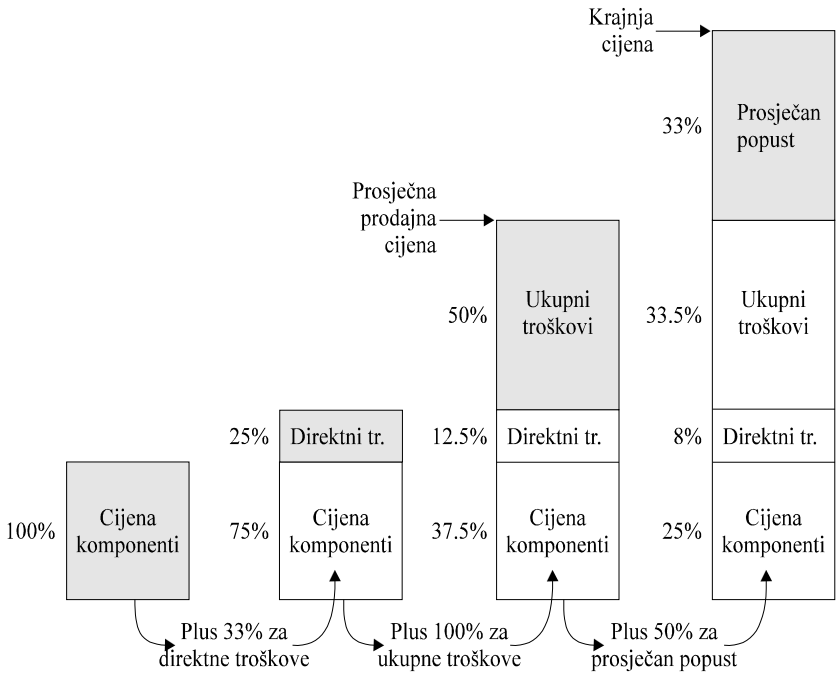


SLIKA 2.10. Cijene više generacija DRAM-komponenti izražene u USA$. Treba imati u vidu inflaciju te valute u periodu 1977 - 1982. 1MB memorije je sa $5000 u 1977 pao na oko $6 u 1995. Svaka generacija na kraju životnoh vijeka vrijedi 8 do 10 puta manje nego na početku. Rast cijene opreme za proizvodnju čini da cijene komponenti ipak lagano rastu. Periodi krize na tržištu su vidljivi (npr. 1987-88 i 1992-93).

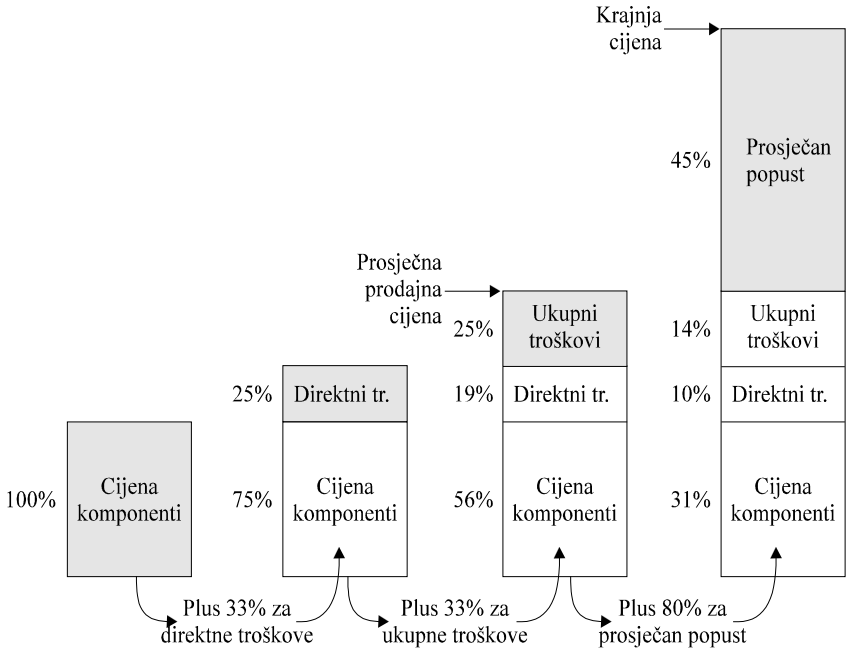
Trend cijena ima važnu ulogu u čitavom opsegu računarskih arhitektura od "superračunara" do personalnih računara. Vrijeme provedeno na tržištu i veličina proizvodnih serija bitno utiču na cijenu proizvoda. "Kriva učenja" u proizvodnji objašnjava pad cijene proizvodnje tokom vremena. Tipičan primjer je procenat ispravnih – "dobitak" (eng. "yeald") kod proizvodnje čipova. Njegovo udvostručenje znači prepolovljenje cijene proizvodnje (slika 2.10.). Masovna proizvodnja čini "krivu učenja" strmijom. To se naročito odnosi na standardne - masovne proizvode računarske industrije (am. "comodities"). Cijena DRAM-e, malih diskova, monitora, tastatura itd. je relativno niska i zbog velike konkurencije (ponuda/potražnja!) na svjetskom tržištu. Iako cijena IC-a stalno pada, njihov relativan udio u cijeni sistema je veoma bitan. Čipovi se proizvode, testiraju i pakuju. Cjena mu se, u osnovi, formira kao zbir ova tri troška podjeljen sa "dobitkom".

Od troškova proizvodnje do cijene proizvoda (slike 2.11 i 2.12.) dug je put. Direktni troškovi predstavljaju troškove proizvodnje, rada, materijala, otpada, garantnog roka i predstavljaju dodatak od 20-40% na cijenu komponenti. Troškovi poslovanja (eng. - gross margin), najčešće, ne mogu teretiti jedan proizvod i odnose se na troškove razvoja, marketinga, prodaje, održavanja opreme, iznajmljivanja prostora, cijenu kapitala, profit prije poreza i porez. Svi ovi troškovi skupa čine srednju prodajnu cijenu (ASP, od eng. - average selling

price). Na tu cijenu se dodaju trgovačke marže od 40-50% i tako se dobije tržišna cijena (eng. - list price). Varijacije u cijeni utiču na prodaju/profit. Kod masovnih proizvoda koje nudi veći broj ponuđača, margina profita je znatno niža.



SLIKA 2.11. Komponente cijene radne stanice srednje klase. Procenti nove cijene za sve elemente su date lijevo od svake koline. (Ilustracija iz sredine 1990-ih godina.)



Slika 2.12. Komponente cijene personalnog računara. (Ilustracija iz sredine 1990-ih godina.)

Mjerenje performansi mora biti precizno definisano kada se kaže da je npr. računar X brži, jači od računara Y. Vrijeme izvršenja nekog zadatka/programa se može različito definisati:

1. ukupno vrijeme,

2. vijeme odziva,
3. kašnjenje - proteklo vrijeme,

računajući vrijeme pristupa disku, memoriji, U/I aktivnosti, administriranje (operativni sistem). Kod multiprogramiranja se stvari još više usložnjavaju - CPU radi nešto drugo dok se čeka na podatke sa diska, pa se prikriva kašnjenje (eng. elapsed time). Zato se uvodi CPU time što isključuje čekanja na U/I i izvršavanje drugih programa. ”CPU time” se dalje dijeli na korisnički (eng. - user CPU time) i sistemski (OS-CPU time) dio. Sistemski dio čini vrijeme koje potroši operativni sistem uslužujući zahtjeve programa.

U skladu s tim je potrebno pažljivo odabrati testni program (eng. benchmark) ili kolekciju programa različitih profila koji će što objektivnije ocijeniti/izmjeriti performanse sistema (tabela 2.5). Pri tome se mora voditi računa da se isti rezultati moraju dobiti u više navrata (bez obzira na trenutne okolnosti). Performanse personalnog računara i radne stanice u višekorisničkom okruženju se mjere različitim specijalizovanim benchmark – programima.

Benchmark	Jezik	Linija koda	Opis
espresso	C	13,500	Minimizira Boole-ove funkcije.
li	C	7,413	Lisp interpreter pisan u C-u, rješava problem 9-kraljica.
eqntott	C	3,376	Prevodi Boole-ove jednačine u tabele istine.
compress	C	1,503	Izvodi kompresiju podataka iz 1-MB fajla koristeći Lempel-Ziv kodiranje.
sc	C	8,116	Izvodi unakrsno tabelarno proračunavanje unutar UNIX tabela.
gcc	C	83,589	Sastoji se od GNU C kompajlera koji konvertuje preprocesirani fajl u optimizirani Sun-3 asemblerski kod.
spice2g6	FORTRAN	18,476	Program za simulaciju manjih elektronskih sklopova.
doduc	FORTRAN	5,334	Monte Carlo simulacija komponenti nuklearnog reaktora.
mdljdp2	FORTRAN	4,458	Program za analize u hemiji, rješava jednsčine kretanja za model od 500 atoma - slično modeliranju strukture tečnog argona.
wave5	FORTRAN	7,628	Dvo-dimenzionalna simulacija electromagnetnih čestica za studiranje razuličitih fenomena plasme. Rješava jednačine kretanja 500 000 čestica u 50 000 tačaka u rešetki u 5 vremenskih koraka.
tomcatv	FORTRAN	195	Program za generisanje mreža - može se lako vektorisati.
ora	FORTRAN	535	Analiza prolaza zrakâ kroz optičke sisteme sfernih i ravnih površina.
mdljsp2	FORTRAN	3,885	Isto kao mdljdp2, ali jednostruke preciznosti.
alvinn	C	272	Simulira trening neuralne mreže. Koristi jednostruku preciznost.
ear	C	4,483	Model srednjeg uha koji filtrira i detektuje različite zvuke i generiše govorne signale. Koristi jednostruku preciznost.
swm256	FORTRAN	487	Model plitke vode koji rješava odgovarajeće jednačine metodom konačnih razlika sa 256×256 mrežom. Koristi jednostruku preciznost.
su2cor	FORTRAN	2,514	Računa mase elementarnih čestica iz Quark-Gluon teorije.
hydro2d	FORTRAN	4,461	Program za astrofiziku koji rješava hidro-dinamičke Navier Stoke jednačine za računanje galaktičkih mlazova.
nasa7	FORTRAN	1,204	Sedam programa za manipulaciju matricama, FFT, Gauss-ovu eliminaciju, kreiranje vrtloga.
fpppp	FORTRAN	2,718	Program za analizu u kvantnoj hemiji, računa integralne derivacije dvaju elektrona.

Tabela 2.5. Programi u sklopu SPEC92 benchmarka. Prvih šest su cjelobrojne operacije i iz njih se računa SPECint92. Ostali su opracije sa pomičnim zarezom (FP-operacije) i određuje SPECfp92.

2.2.1. Kvantitativni principi dizajna računara

Prilikom pravljenja kompromisa u dizajnu računarske arhitekture, cilj je više obratiti pažnje na ono što se češće dešava (ponavlja) nego na rijetke slučajeve.

Primjer: ako se kod sabiranja rijetko dešava prekoračenje, koji slučaj optimizirati pa da

ukupne performanse sistema porastu?

Taj problem tretira Amdhal-ov zakon koji kaže: "*Povećanje performansi korištenjem nekog ubrzanja je ograničeno dijelom vremena kada se ono može koristiti.*"

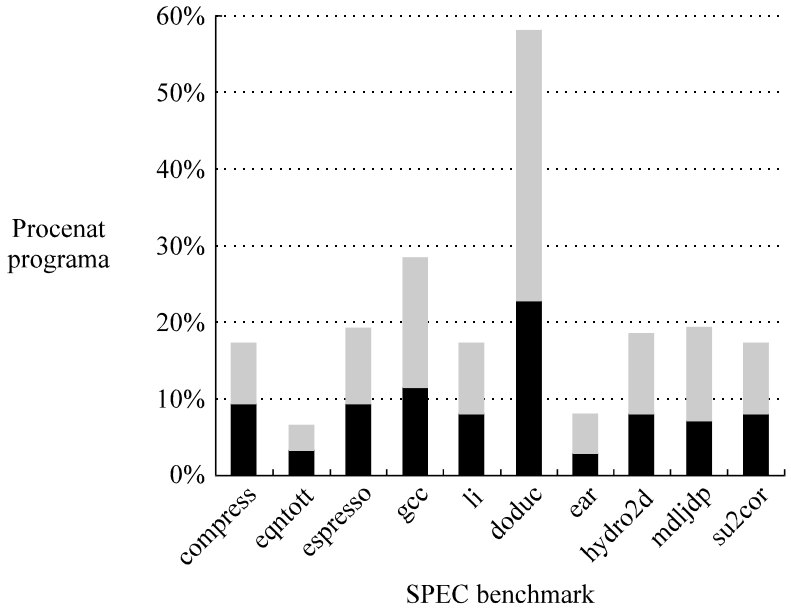
Zato se nameću pitanja:

1. koliko je ubrzanje i
2. koliko vremena se koristi (dolazi do izražaja)?

Parametri performansi su sljedeći:

1. period signala sata (eng. - clock cycle time) - zavisi od hardverske tehnologije i organizacije,
2. broj ciklusa sata po instrukciji (CPI) - zavisi od organizacije i arhitekture instrukcijskog seta (ISA),
3. broj izvršenih instrukcija (instruction count) - zavisi od ISA-e i tehnologije kompajlera.

Sve ovo je međusobno usko povezano, pa se postavlja pitanje kako mjeriti performanse a pri tome ne izgubiti iz vida faktore kao što su rezidentni programi (i operativni sistem), stanje u CACHE-evima (i diskova), osobine programa koji se izvršavaju. Lokalnost referenciranja (u memoriji i na disku) dovodi do toga da se "90% izvršenjâ izvršava 10% instrukcija" (slika 2.13.).



Slika 2.13. Dijagram pokazuje koji procenat instrukcija se izvršava u 80% (tamni dio) i 90% (svjetliji dio) izvršenjâ instrukcija. Npr. u compress programu oko 9% instrukcija se izvršava u 80% a oko 16% u 90% izvršenjâ instrukcija. Prosječno 90% izvršenjâ dolazi od 10% instrukcija kod cjelobrojnih i oko 14% u operacijama sa pomočnim zarezom.

Sa svakom novom generacijom računara zastarjevaju metode mjerenja performansi od prethodne. MIPS-i (milioni instrukcija po sekundi) i MFLOPS-i (milioni operacija sa pokretnim zarezom po sekundi) nisu objektivni pokazatelji, već više marketinški trikovi proizvođača.

3. Arhitektura skupa instrukcija

Po tome kako se unutar procesora (CPU-a) smještaju podaci, računarske arhitekture se mogu dijeliti na:

1. stack-orijentisane (0-adresne),
2. akumulatorske,
3. registarske - one sa registrima opšte namjene,
4. memorijske.

Kod stack-orijentisanih, operandi se implicitno nalaze na vrhu stack-a. Kod akumulatorskih, jedan operand je implicitno u akumulatoru. Registarske arhitekture imaju samo eksplicitne operande - bilo direktno iz memorije (reg/mem) ili posredno preko privremenih registara (load/store arhitekture). Memorijske arhitekture drže sve operande u memoriji. One su bile aktuelne u prvim generacijama digitalnih računara, ali se više ne koriste.

Kao ilustracija razlika između prve tri vrste arhitektura može poslužiti primjer kako svaka izvršava operaciju C=A+B.

STACK	AKUMULATOR	REG/MEM	REG(LOAD/STORE)
PUSH A	LOAD A	LOAD R1, A	LOAD R1, A
PUSH B	ADD B	ADD R1, B	LOAD R2, B
ADD	STORE C	STORE C, R1	ADD R3, R1, R2
POP C			STORE C, R3

Tabela 3.1 Načini na koje tri različite arhitekture izvršavaju operaciju C=A+B.

Registri opšte namjene su efikasni iz dva osnovna razloga:

1. brži su od vanjske memorije,
2. mogu se efikasnije koristiti (kompajlerski!) od drugih tipova arhitektura.

Izraz (A*B)-(C*D)-(E*F) se može odraditi množeći bilo kojim redoslijedom, što može biti efikasnije zbog položaja operanada ili problema sa protokom kroz protočnu strukturu (pipeline). Na stack-mašinama bi se to moralo odrađivati slijeva nadesno ili bi se morale raditi zamjene operanada na stack-u.

Držanjem varijabli u registrima smanjuje se saobraćaj sa memorijom, ubrzava izvršavanje programa, kodiranje instrukcija se poboljšava (potrebno je manje bita za referenciranje podataka u registrima nego u memoriji).

Kompromisi se prave sa registrima specifične namjene. Koliko registara je potrebno - zavisi koliko ih kompajleri koriste. Oni rezervišu jedan dio registara za odrađivanje aritmetičkih operacija (eng. expression evaluation), drugi dio za prenos parametara a ostatak za držanje varijabli.

Arhitekture sa registrima opšte namjene se dijele na dvije glavne grupe, u zavisnosti od prirode operanada za tipičnu ALU-operaciju

1 ALU-instrukcija ima 2 ili 3 operanda (sa 2, jedan izvor je i odredište),
2 koliko operanada se može adresirati u memoriji (0-3 tipično)

Prednosti i mane zavise od kompajlera i načina realizacije. Najvažniji uticaj odabrane

konfiguracije je u kodiranju instrukcija i broju instrukcija potrebnih za izvršenje nekog zadatka.

3.1. Načini adresiranja memorije

Može se adresirati bajt, poluriječ (16 bita), riječ (32) i dvostruka riječ (64). Obzirom na redoslijed smještanja bajta riječi u memoriji postoji:

1. **Little Endian** - adresa xxxxxx00 pokazuje na najniži bajt u riječi (adresa podatka je adresa najmanje značajnog bajta).
2. **Big Endian** - adresa xxxxxx00 pokazuje na najznačajniji (Big) bajt u riječi (adresa podatka je adresa najznačajnijeg bajta).

Ova različitost pretstavlja problem kod razmjene podataka među mašinama sa različitim redoslijedom smještaja bajta.

Za pristup podacima većim od jednog bajta može se pojaviti problem poravnanja (eng. alignment). Za S-bajtni pristup na bajtnoj adresi A se kaže da je poravnat ako je $A_{mod}S=0$, iz čega se mogu dobiti podaci iz sljedeće tabele.

S	poravnat na bajtnoj adresi	neporavnat na bajtnoj adresi
1 – bajt	0, 1, 2, 3, 4, 5, 6, 7	NIKADA
2 – poluriječ	0, 2, 4, 6	1, 3, 5, 7
4 – riječ	0, 4	1, 2, 3, 5, 6, 7
8 – poluriječ	0	1, 2, 3, 4, 5, 6, 7

Tabela 3.2 Poravnanja različitih formata podataka u memoriji

Zašto je neporavnat pristup problem? Komplikuje hardver jer je memorija obično složena po riječima ili duplim riječima. Neporavnat pristup se svodi na više poravnatih uz dodatno sortiranje bajta. Kod poravnatog pristupa, pristup pojedinim bajtima/riječima zahtijeva dodatan hardver za poravnavanje bajta i poluriječi u registru. Zatim se obično koriste instrukcije za proširenje predznakom (2kk), ili se zahtijeva da one ne utiču na ostale bite u registru. Kod upisa je potrebno mijenjati samo izabrani bajt (što se može odraditi sa više pristupa i dodatnim logičkim operacijama).
Kod arhitektura sa registrima opšte namjene adresiranjem se može specificirati konstanta, registar ili memorijska lokacija. Pristup memoriji je definisan efektivnom adresom.

Najčešće korišteni načini adresiranja kod pristupa podacima su dati u tabeli 3.3. Adresiranja koja zavise od stanja programskog brojača (PC) su ispuštena.

Načini adresiranja imaju sposobnost da:

1 značajno smanje broj instrukcija,
2 povećaju složenost mašine i
3 mogu povećati prosječan broj ciklusa sata po instrukciji (CPI).

Način adresiranja	Primjer instrukcije	Značenje	Kada se koristi
Registarsko	Add R4, R3	Regs[R4]←Regs[R4] +Regs[R3]	Kada je vrijednost u registru.
Nepostredno	Add R4, #3	Regs[R4]←Regs[R4]+3	Za konstante.
Relativno	Add R4, 100(R1)	Regs[R4]←Regs[R4] +Mem[100+Regs[R1]]	Za pristup lokalnim varijablama.

Registarsko indirektno	Add R4, (R1)	Regs[R4]←Regs[R4] +Mem[Regs[R1]]	Za pristup pokazivačima ili izračunatim adresama.
Indeksirano	Add R3, (R1+R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1] +Regs[R2]]	Kod adresiranja nizova: R1=početak niza; R2=index.
Direktno ili apsolutno	Add R1, (1001)	Regs[R1]←Regs[R1] +Mem[1001]	Kod pristupa statičkih podataka; može biti potrebna velika adresna konstanta .
Memorijsko indirektno	Add R1, @(R3)	Regs[R1]←Regs[R1] +Mem[Mem[Regs[R3]]]	Ako je R3 adresa pokazivača *p*, pristupa se **p*.
Autoinkrement	Add R1, (R2)+	Regs[R1]←Regs[R1] +Mem[Regs[R2]] Regs[R2]←Regs[R2]+*d*	Za pristupanje nizovima u petlji. R2=početak niza; svaki pristup inkrementira R2 za veličinu elementa, *d*.
Autodekrement	Add R1, -(R2)	Regs[R2]←Regs[R2]-*d* Regs[R1]←Regs[R1] +Mem[Regs[R2]]	Isto kao autoinkrement. Autodekrement/inkrement se može koristiti kao push/pop za realizaciju steka.
Skalirano	Add R1, 100(R2) [R3]	Regs[R1]←Regs[R1] +Mem[100+Regs[R2] +Regs[R3]**d*]	Za indeksiranje nizova. Može se primijeniti u svim indeksiranim adresiranjima kod nekih mašina.

Tabela 3.3 Izbor načina adresiranja sa primjerima, značenjem i korištenjem. Variabla d označava veličinu podataka kojim se pristupa (1, 2, 4 ili 8 bajta).

3.2. Operacije u skupu instrukcija

Operacije koje podržava većina savremenih (Instruction Set Architecture - ISA) arhitektura su date u tabeli 3.4. Opšte je pravilo u svim ISA-ma da se najčešće izvršavaju jednostavne instrukcije.

Tip operatora	Primjeri
Aritmetički i logički	Cjelobrojne aritmetičke i logičke operacije: dodaj, "I", oduzmi, "ILI"
Prenos podataka	Čitanja-Pisanja (instrukcije prebacivanja na mašinama sa adresiranjem memorije)
Upravljački	Grananje, skok, poziv procedure i povratak, trap
Sistemski	Pozivi operatvnog sistema, instrukcije za upravljanje virtuelnom memorijom
Pomični zarez	Operacije sa pomičnim zarezom: dodaj, pomnoži
Decimalni	Decimalno sabiranje, decimalno množenje, pretvaranje decimalno-u-karakter
String	Premještanje stringova, poređenje stringova, pretraživanje stringova
Grafički	Operacije nad pikselima, operacije kompresije/dekompresije

Tabela 3.4 Kategorije operatora u instrukcijama i njihovi primjeri.

Tabela 3.5 pokazuje da je 96% izvršenih instrukcija u SPECint92 na 80x86 navedenih 10 instrukcija. Prema tome posebnu pažnju bi trebalo posvetiti implementaciji takvih instrukcija (i tim redom - Amdhal-ov zakon).

Rang	80x86 instrukcija	Cjelobrojno prosjek
1	čitanje iz memorije	22%
2	uslovno grananje	20%
3	poređenje	16%
4	upis u memoriju	12%

5	sabiranje	8%
6	logičko "I"	6%
7	oduzimanje	5%
8	premještanjeregistar-u-register	4%
9	poziv podprograma	1%
10	povratak iz podprograma	1%
	Ukupno	96%

Tabela 3.5 Deset najčešće izvršavanih instrukcija i80x86 u SPECint92 programima.

3.2.1. Upravljačke instrukcije

Upravljačke instrukcije se obično dijele na skokove (bezuslovne) i grananja (uslovna). Oba mogu biti apsolutna i relativna (relokatibilnost!). Detaljnija je podjela na:

1. uslovno grananje,
2. skokove,
3. poziv procedura,
4. povratak iz procedura.

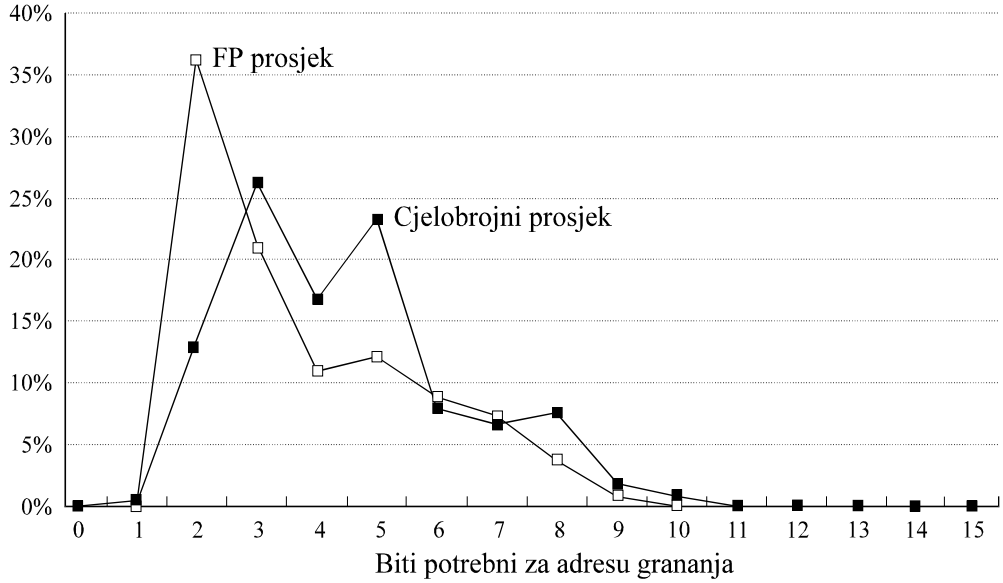
Odredišna adresa uvjek mora biti definisana. Izuzetak je povratak iz procedura, kada se adresa ne zna u trenutku kompajliranja. Najčešće se ove adrese određuju relativno u odnosu na PC (prog. brojač). Tako se štede biti adrese jer se često radi o bliskim skokovima/grananjima. Na taj način i program postaje relokatibilan - nezavisan od položaja u memorijskom prostoru.

Za realizaciju povratka iz procedure i indirektnih skokova (kod kojih odredišna adresa nije poznata u vrijeme kompajliranja) koristi se dinamičko određivanje adrese - promjenjivo u trenutku izvršavanja. To može biti registarsko indirektno adresiranje.

Takvi skokovi se mogu koristiti za:

1 CASE ili SWITCH izraze u jezicima visokog nivoa,
2 dinamički dijeljenim bibliotekama (koje se učitavaju samo po potrebi),
3 virtuelne funkcije (kod objektno orjentisanih jezika npr. C++, omogućava pozive različitih rutina u zavisnosti od tipa podataka koji se koriste).

Grananja koriste PC-relativno adresiranje. Osnovno je pitanje koliko bita je neophodno izdvojiti za displacement (koliko su daleke odredišne adrese). Slika 3.1. daje jednu statističku raspodjelu broja tih bita. Vidi se da se grananja vrše ± 100 instrukcija, pa je za displacement potrebno najmanje 8 bita.



Slika 3.1. Udaljenost grananja u broju instrukcija između instrukcije grananja i odredišne instrukcije. Odredišta su kod cjelobrojnih operacija najčešće četiri do sedam instrukcija daleko (2 do 3 bita) od grananja.

Jedna od važnijih osobina grananja je da je veliki broj poređenja jednostavan test jednakosti/nejednakosti i veliki broj poređenja sa 0 (više od 50% cjelobrojnih poređenja kod grananja je test jednakosti sa 0).

Pozivi procedura i povratci sadrže prenos kontrole (skok/grananje) i smještaj stanja (konteksta - ako ništa drugo onda povratnu adresu). Neke arhitekture obezbjeđuju mehanizme za sačuvavanje sadržaja registara, dok druge to očekuju od kompajlera. Registre može sačuvati pozivajuća ili pozvana procedura - zavisno od pristupa globalnim varijablama i poziva procedura (međusobnih, višestrukih, ugnježdavanja). Složeni kompajleri koriste kombinaciju ove dvije metode.

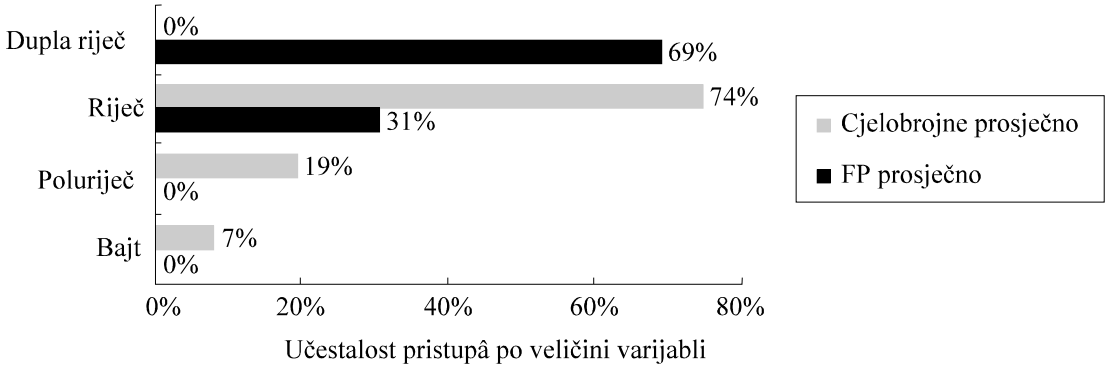
3.2.2. Tipovi i veličine operanada

Najčešće se tip operanda određuje kodiranjem u kod operacije (eng. OPCODE). Alternativno, oznake se stavljaju uz operande i interpretiraju se hardverski i u skladu s tim se izabiraju operacije. Takve se arhitekture zovu "Tagged-data-machines" i danas se više ne koriste.

Tip operanda, u glavnom, određuje i njegovu veličinu:

1. **logički** - 1 bit,
2. **karakter** (1 bajt) su najčešće u ASCII-u, ponekad BCD (zapakovani ili raspakovani),
3. **poluriječ** (2 bajta), UNICODE -"16 bitni ASCII" i
4. **riječ** (4 bajta) se kao cjelobrojne vrijednosti kodiraju u 2KK,
5. **riječ kao broj sa pomičnim zarezom** - jednostruke preciznosti i
6. **dupla riječ (8 bajta) kao broj sa pomičnim zarezom** - dvostruke preciznosti (prema IEEE 754 standardu).

Distribucija veličine podataka kojima se pristupa u memoriji (slika 3.2.) pomaže u odlučivanju koje tipove podataka najefikasnije podržavati.



Slika 3.2. Raspodjela učestanosti pristupa podacima po veličini podataka benchmark programima.

3.2.3. Kodiranje instrukcija

Sve što je ranije navedeno utiče na način kodiranja instrukcija. Od kodiranja dalje zavisi:

1. veličina programa i
2. izvedba CPU-a.

Tu se javljaju dva osnovna problema:

1. kako kodirati kod operacije i
2. kako kodirati načine adresiranja.

Ako se odluči ugrađivati velik broj kombinacija broja potrebnih operanada i načina adresiranja, potrebno je uvesti posebno polje adresnog specifikatora za svaki operand. Svako polje određuje način pristupa operandu. S druge strane, kod load/store arhitektura, sa jednim memorijskim operandom i 1-2 načina adresiranja, način adresiranja se može ukodirati direktno u kod operacije.

Najčešće se potroši više bita za kodiranje načina adresiranja i polja za adresiranje registara nego za kod operacije.

Zato treba uravnotežiti sljedeće protivrječnosti:

1. želja za što više registara i načina adresiranja,
2. uticaj veličine polja za definisanje registara i načina adresiranja na prosječnu veličinu instrukcije - programa,
3. želja da instrukcije budu one veličine koja je pogodna za obradu i implementaciju. Da li će instrukcije biti dužine 1, 2, 3 ili više bajta i biti “kompaktne” ali “teške” za izvršavanje ili će biti fiksne dužine i “lake” za implementaciju ali često “glomazne”.

U skladu s tim postoje tri načina kodiranja:

1. **varijabilni** - koji omogućava sve kombinacije operacija i načina adresiranja (npr. VAX arhitektura sa instrukcijama dužine do 56 bajta!),
2. **fiksni** - kod kojega se operacija i način adresiranja kodiraju u jedinstveni kod operacije i često su sve instrukcije iste veličine (npr. MIPS, PowerPC, HP-PA, SPARC),
3. **hibridni** - kompromisni između varijabilnog i fiksnog i, ujedno, između veličine programa i lakoće dekodiranja u CPU-u (npr. INTEL 80x86). Ovim se nastoje smanjiti različitosti formata instrukcija i, istovremeno, omogućiti više različitih formata u cilju smanjenja ukupne veličine kôda.

Biranjem između ovih načina kodiranja, bira se između kompaktnog kôda (varijabilno kodiranje) i performansi (fiksno kodiranje).

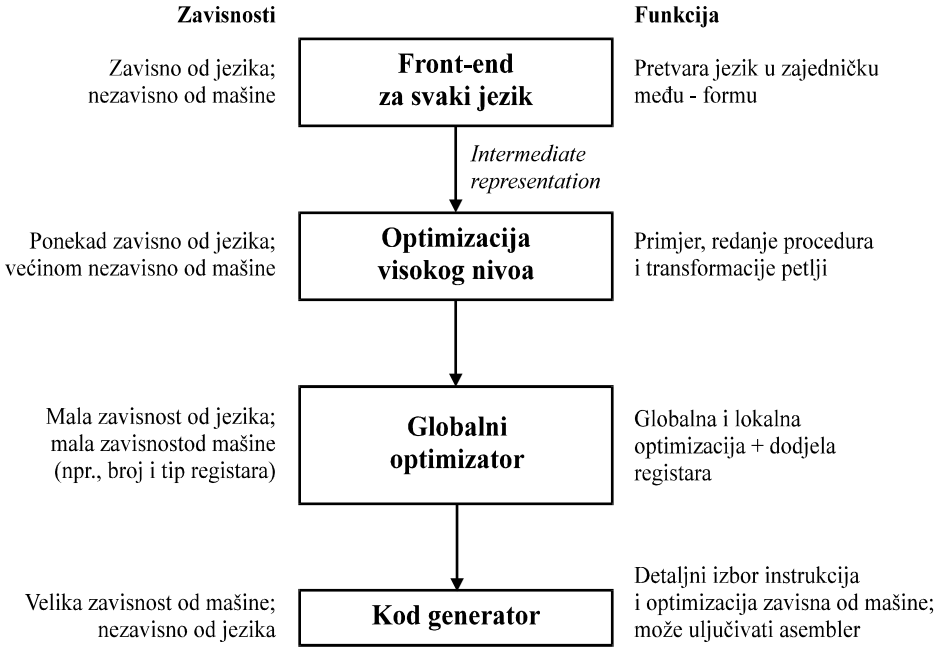
3.3. Uloga kompajlera

Kod savremenih arhitektura skupa instrukcija, projektovanje hardvera i kompajlera su vrlo usko povezani (slika 3.3.). Ciljevi pisca kompajlera su:

1. korektnost - svi ispravni programi se moraju korektno kompajlirati,
2. brzina generisanog koda,

a kao manje važni slijede:

1. brzina kompajliranja,
2. podrška debugger-u,
3. veza sa drugim jezicima visokog nivoa, itd.



Slika 3.3. Današnji kompajleri rade obično u dva do čeriti prolaza.

Svaka faza/prolaz kompajliranja pretvara kôd iz višeg u niži nivo apstrakcije - sve do skupa instrukcija u mašinskom kodu. Kompleksnost kompajlera i njegova 2, 3 ili više prolaza ograničavaju nivo optimizacije. Ona se vrši po nivoima i nema povratnih sprega među njima (problem phase ordering-a).

Optimizacije se dijele na:

1. **optimizacije visokog nivoa** - one kod kojih je ulaz izvorni kôd a izlaz sljedeća faza optimizacije,
2. **lokalna optimizacija** - nad osnovnim blokovima, izravnim skupovima linija bez grananja,
3. **globalna optimizacija** - proširuje lokalnu kroz optimizaciju grananja i petlji,
4. **alociranje registara**,
5. **optimizacija zavisna od konkretne mašine** (arhitekture skupa instrukcija) - detaljno iskorištava poznavanje arhitekture.

Alokacija registara je jedna od najvažnijih optimizacija u kompajlerima. Ostali tipovi optimizacija su dati u tabeli 3.6. (NM - nije mjerljivo)

Optimizacija	Objašnjenje	Procenat od ukupnog broja optimizacionih transformacija
Visokog nivoa	**Na nivou ili blizu izvornog koda; ne zavisi od mašine**	
Integracija procedura	Zamjenjuje pozive procedure tijelom procedure	N.M.
Lokalna	**Unutar osnovnih blokova koda**	
Eliminacija dupliciranja	Zamjenjuje dvije instance istog računanja jednom kopijom	18%
Prosljeđivanje konstanti	Zamjenjuje sve instance dodjeljene varijable tom konstantom	22%
Reduciranje visine steka	Rearanžira izraze/iskaze da bi se minimizirali resursi potrebni za njihovu obradu	N.M.
Globalna	**Izvan osnovnih blokova - preko grananja**	
Globalna eliminacija dupliciranja	Isto kao kod lokalne, ali sada i preko grananja	13%
Prosljeđivanje kopija	Zamjenjuje sve instance variable A kojoj je dodjeljena vrijednostX (tj., A=X) sa X	11%
Premještanje koda	Uklanja kod iz petlje koja računa istu vrijednost u svakoj iteraciji	16%
Eliminacija indukcije variabli	Pojednostavljuje/eliminiše računanjeadresa niza unutar petlji	2%
Zavisna od mašine	**Zavisi od poznavanja mašine**	
Redukcija složenosti	Primjer, zamjenjuje množenje konstantom sabiranjima i šiftanjima	N.M.
Raspoređivanje u PS-i	Mijenjanje redoslijeda instrukcija radi povećanja performansi PS-e	N.M.
Optimizacija daljine grananja	Biranje najkraćeg grananja koje dostiže odredište	N.M.

Tabela 3.6. Glavni tipovi optimizacija sa primjerima.

3.3.1. Uticaj kompajlera na arhitekturu

Kompajleri određuju kako će program, pisan u jeziku visokog nivoa, koristiti arhitekturu skupa instrukcija:

1. kako se alociraju i adresiraju varijable?
2. koliko je registara potrebno za alociranje varijabli?

Postoje 3 područja u kojima jezici visokog nivoa drže varijable:

1. **STACK** - za lokalne varijable, uglavnom skalarne (pojedinačne) a ne nizove,
2. **područje globalnih podataka** - statički deklarisani objekti/globalne varijable i konstante (najčešće nizovi ili skupovi podataka - strukture),
3. **HEAP** - za dinamičke objekte nevezane za stack discipline (pristupa im se pokazivačima i obično su skalari.

Za šta se mogu alocirati registri?

1. za STACK
2. za globalne podatke – teško!
3. za HEAP - nemoguće (jer im se pristupa pokazivačima)

Kako arhitekta može pomoći piscu kompajlera?

Većina današnjih programa su “lokalno jednostavni” i na tom nivou je prevođenje jednostavno. Ali, programi su veliki i “globalno kompleksni” u svojoj interakciji. Neke osobine skupa instrukcija pomažu piscu kompajlera:

1. Regularnost - operacije, tipovi podataka i načini adresiranja trebaju biti ortogonalni (međusobno nezavisni) kada god to ima smisla (npr. da se svi načini adresiranja primjenjuju na svim instrukcijama za prenos podataka).
2. Obezbijediti sredstva za rješenje problema, a ne rješenja. Pokušaj podrške jezicima visokog nivoa može podržati jedan ali ne i druge.
3. Pojednostaviti kompromise među alternativnim rješenjima (ovo se posebno komplikuje sa keševima i protočnim strukturama (npr. koliko puta treba pristupiti varijabli u memoriji da bi se više isplatilo držati je u registru - to je teško izračunati i varira).
4. Obezbijediti instrukcije koje će vrijednosti poznate u trenutku kompajliranja smatrati konstantama - vrijednost poznatu tada ne treba ponovo računati u trenutku izvršenja.

Zato se kaže “manje je više” - u dizajnu skupa instrukcija!

4. Arhitektura oglednog procesora

Osnovne karakteristike arhitekture oglednog procesora su definisane brojem i namjenom registara, tipovima podataka koji će se obrađivati i podržanim načinima adresiranja. Sve troje bitno utiče na njegovu složenost i performanse.
Neka je skup registara definisan kao

1. 32 32-bitna registra opšte namjene R0 - R31 i
2. 32 32-bitna registra za operacije sa pomičnim zarezom (FP od eng. Floating Point) jednostruke preciznosti ili 16 64-bintna FP registra dvostruke preciznosti F0, F2, ..., F30.

Pri tome R0=0 - ožičena vrijednost.

Tipovi podataka koje procesor podržava su

1. 1, 2 i 4 bajta za cjelobrojne vrijednosti, (zbog postojeće popularnosti u jezicima visokog nivoa, kao što je C),
2. 4 ili 8 bajta za FP jednostruke/dvostruke preciznosti, operacije rade nad 32-bitnim cjelobrojnim, 32-bitnim i 64-bitnim FP-vrijednostima, bajti i poluriječi se upisuju u registre - znakom prošire (2-komplement kod) i koriste kao 32-bitne vrijednosti.

Odabrani načini adresiranja su:

1. neposredno (za konstante, npr. Add R4, #3) i
2. relativno (za lokalne varijable, npr. Add R4, 100(R1)).

Oba sa 16-bitnim poljima, a od njih se izvode registarsko indirektno (kada je displacement jednak 0) i apsolutno (kada je R0 uzet za bazni registar). Prema tome, postoje 4 načina adresiranja iako su samo 2 podržana arhitekturom.

Memorija je bajt-adresabilna i organizovana po Big Endian principu sa 32-bitnom adresom. Svi pristupi memoriji su tipa čitaj/piši (eng. Load/Store) u i iz cjelobrojnih registara opšete namjene ili FP i moraju biti poravnati.

4.1. Format instrukcija

Za programski model oglednog procesora odabrana su dva načina adresiranja, pa je to efikasno ukodirati zajedno sa kodom operacije. Zbog lakoće realizacije protočne strukture i dekodiranja sve instrukcije su 32-bitne sa 6 bita osnovnog koda operacije (slika 4.1.) i 16-bitnim poljem za relativno adresiranje, neposredne konstante i PC-relativno grananje.

I - tip instrukcije

6	5	5	16
Opkod	rs1	rd	Neposredna vrijednost

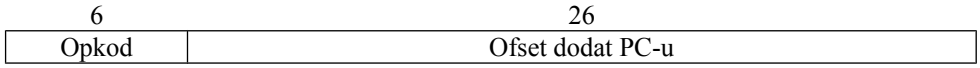
Kodira: Memorijski pristup bajtima, riječima, poluriječima
Sve neposredne vrijednosti (rd ← rs1 op neposredna vrijednost)
Uslovna grananja (rs1 je registar, rd se ne koristi)
Skok na registar, skok i link na registar (rd=0, rs=odredište, neposredna vrijednost=0)

R - tip instrukcije

6	5	5	5	11
Opkod	rs1	rs2	rd	funkcija

Registar-registar ALU operacija: rd ← rs1 funk rs2
Funkcija kodira operaciju na putu podataka: Dodaj, Oduzmi, ...
Čitanje/pisanje specialnih registara i premještanja

J - tip instrukcije



Skok i skok sa linkom
Trap i povratak iz izuzetka

Slika 4.1. Raspored polja za tri vrste instrukcija. Sve instrukcije su kodirane kao jedan od ova tri tipa.

Operacije:

1. Load/Store (pri čemu upis u R0 nema smisla)
2. ALU-operacije
3. grananje i skokovi
4. FP operacije

Load/Store se odnosi na sve cjelobrojne registre opšte namjene (osim upisa u R0) i sve registre za rad sa pokretnim zarezom, bilo jednostruke ili dvostruke preciznosti. Konverzija jednostruka - dvostruka preciznost se mora odrađivati eksplicitno (po IEEE 754 standardu). Primjeri su dati u tabeli 4.1., a značenja nekih oznaka su:

- $\leftarrow_n$ - znači transfer n-bitne vrijednosti,
- $Regs[R4]_0$ - pretstavlja predznak u registru R4 a $Regs[R3]_{24..31}$ - donji bajtovi u R3 (najznačajniji bit je na poziciji 0),
- 0^{24} - pretstavlja polje nula dužine 24 bita,
- ## znači povezivanje dva polja (eng. concatenate).

Tako, na primjer:

$$Regs[R10]_{16...32} \leftarrow (Mem[Regs[R8]]_0)^8 \,\#\#\, Mem[Regs[R8]]$$

znači bajt na memorijskoj lokaciji određenoj sadržajem R8 je proširena predznakom do 16 bita i smješten u donjih 16 bita R10 (gornjih 16 ostaju netaknuti).

Primjer instrukcije	Ime Instrukcije	Značenje
LW R1, 30(R2)	Load word	$Regs[R1] \leftarrow_{32} Mem[30+Regs[R2]]$
LW R1, 1000(R0)	Load word	$Regs[R1] \leftarrow_{32} Mem[1000+0]$
LB R1, 40(R3)	Load byte	$Regs[R1] \leftarrow_{32} (Mem[40+Regs[R3]]_0)^{24}$ ##$Mem[40+Regs[R3]]$
LBU R1, 40(R3)	Load byte unsigned	$Regs[R1] \leftarrow_{32} 0^{24}$##$Mem[40+Regs[R3]]$
LH R1, 40(R3)	Load half word	$Regs[R1] \leftarrow_{32} (Mem[40+Regs[R3]]_0)^{16}$ ##$Mem[40+Regs[R3]]$##$Mem[41+Regs[R3]]$
LF F0, 50(R3)	Load float	$Regs[F0] \leftarrow_{32} Mem[50+Regs[R3]]$
LD F0, 50(R3)	Load double	$Regs[F0]$##$Regs[F1] \leftarrow_{64} Mem[50+Regs[R3]]$
SW 500(R4), R3	Store word	$Mem[500+Regs[R4]] \leftarrow_{32} Regs[R3]$
SF 40(R3), F0	Store float	$Mem[40+Regs[R3]] \leftarrow_{32} Regs[F0]$
SD 40(R3), F0	Store double	$Mem[40+Regs[R3]] \leftarrow_{32} Regs[F0]$; $Mem[44+Regs[R3]] \leftarrow_{32} Regs[F1]$
SH 502(R2), R3	Store half	$Mem[502+Regs[R2]] \leftarrow_{16} Regs[R3]_{16..31}$
SB 41(R3), R2	Store byte	$Mem[41+Regs[R3]] \leftarrow_{8} Regs[R2]_{24..31}$

Tabela 4.1. Instrukcije čitanja i pisanja u memoriju. Sve koriste jedan način adresiranja - relativno i očekuju da je vrijednost u memoriji poravnata. Obje vrste instrukcija podržavaju sve tipove podataka.

(2) ALU-operacije su registarsko-registarske i podrazumjevaju +, -, AND, OR, XOR, SHIFT. Punjenje konstante (neposredni - LI) je sabiranje sa [R0] kao i transfer među registrima (tabela 4.2.).

Primjer instrukcije	Ime instrukcije	Značenje
ADD R1, R2, R3	Add	Regs[R1]←Regs[R2]+Regs[R3]
ADDI R1, R2, #3	Add immediate	Regs[R1]←Regs[R2]+3
LHI R1, #42	Load high immediate	Regs[R1]←42##0^{16}
SLLI R1, R2, #5	Shift left logical immediate	Regs[R1]←Regs[R2]<<5
SLT R1, R2, R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

Tabela 4.2. Primjeri aritmetičkih i logičkih instrukcija sa i bez neposrednih vrijednosti.

(3) instrukcije za poređenje registara (=, ≠, <, >, ≤, ≥). Ako je uslov ispunjen, instrukcija upisuje 1 u odredišni registar, inače upisuje 0.

Postoje dva načina skakanja:

1. klasični sa upisom nove lokacije u PC i
2. skok sa linkovanjem (ostavlja povratnu adresu u R31) za pozive procedura.

Grananja su uslovna. Uslov je dat u instrukciji i može se testirati izvorišni registar na jednakost ili nejednakost sa nulom, pri čemu registar može sadržati vrijednost ili rezultat poređenja. Odredišna adresa grananja se određuje dodavanjem 16-bitnog ofseta na PC (koji pokazuje na sljedeću instrukciju). Tabela 4.3. daje prikaz tipičih instrukcija skokova i grananja (uključujući i test FP-status registra tipa tačno/netačno).

Primjer instrukcije	ime instrukcije	Značenje
J name	Jump	PC←name; $((PC+4)-2^{25})\leq name<((PC+4)+2^{25})$
JAL name	Jump and link	Regs[R31]←PC+4; PC←name; $((PC+4)-2^{25})\leq name<((PC+4)+2^{25})$
JALR R2	Jump and link register	Regs[R31]←PC+4; PC←Regs[R2]
JR R3	Jump register	PC←Regs[R3]
BEQZ R4, name	Branch equal zero	if (Regs[R4]==0) PC←name; $((PC+4)-2^{15})\leq name<((PC+4)+2^{15})$
BNEZ R4, name	Branch not equal zero	if (Regs[R4]!=0) PC←name; $((PC+4)-2^{15})\leq name<((PC+4)+2^{15})$

Tabela 4.3. Tipične instrukcije za upravljanje tokom programa. Sve osim skoka na adresu iz registra su PC-relativne.

(4) FP-instrukcije manipulišu FP-registrima uz indikaciju F-jednostruku i D-dvostruku preciznost. Postoje i instrukcije za prenošenje u/iz FP i cjelobrojnih registara i to jednostruke preciznosti u jednoj a dvostruke u dvije instrukcije. Dodate su instrukcije za cjelobrojno množenje i dijeljenje nad 32-bitnim FP-registrima kao i konverzija cjelobrojnih u FP i obratno. FP-operacije su +, -, * i / za D i F sufikse. FP-poređenje setuje bit u posebnom FP-status registru koji testiraju PFPT i PFPF (true/false) instrukcije.

Neobična osobina ogledne arhitekture je to da koristi FPU za cjelobrojno množenje i dijeljenje jer je FPU svakako složenija od one za cjelobrojne operacije. Zato operande treba držati u FP-registrima. Tabela 4.4. daje opis svih instrukcija ogledne arhitekture.

Tip/opkod instrukcije	Značenje instrukcije
Prenos podataka	Premješta podatke između registara i memorije, ili između cjelobrojnih i FP ili specialnih registara; jedini način pristupa memoriji je 16-bitni displacement+sadržaj cjelobrojnog registra opšte namjene (GPR, od eng. General Purpose Register).
LB, LBU, SB	Load byte, load byte unsigned, store byte
LH, LHU, SH	Load half word, load half word unsigned, store half word
LW, SW	Load word, store word (u/iz cjelobrojnih registara)
LF, LD, SF, SD	Load SP float, load DP float, store SP float, store DP float
MOVI2S, MOVS2I	Move iz/u GPR u/iz specialnog registra
MOVF, MOVD	Kopiranje FP registra ili DP para u drugi registar ili par
MOVFP2I, MOVI2FP	Move 32 bits iz/u FP registre u/iz cjelobrojnih registara
Aritmetička/logička	Operacije nad cjelobrojnim ili logičkim podacima u GPR, aritmetiku sa predznakom trap prekoračenje
ADD, ADDI, ADDU, ADDUI	Add, add immediate (16 bits); sa i bez predznaka
SUB, SUBI, SUBU, SUBUI	Subtract, subtract immediate; sa i bez predznaka
MULT, MULTU, DIV, DIVU	Multiply and divide, sa i bez predznaka; operandi moraju biti u FP registrima; sve operacije uzimaju i daju 32 bitne vrijednosti
AND, ANDI	And, and neposredno
OR, ORI, XOR, XORI	Or, or neposredno, exclusive or, exclusive or neposredno
LHI	Load high immediate; upisuje gornju polovinu registra sa neposredno
SLL, SRL, SRA, SLLI, SRLI, SRAI	Shifts: neposredno (SI) i variabilno iz (S); šift lijevo logički, desno logički, desno aritmetički
S__, S__I	Set conditional: "__" može biti LT, GT, LE, GE, EQ, NE
Upravljačka	Uslovna grananja i skokovi; PC-relativni ili kroz registar
BEQZ, BNEZ	Branch GPR equal/not equal to zero; 16 bit offset u odnosu na PC+4
BFPT, BFPF	Test comparison bit in the FP status register and branch; 16 bit offset u odnosu na PC+4
J, JR	Jumps: 26 bitni offset u odnosu na PC+4 (J) ili odredište u registru (JR)
JAL, JALR	Jump and link: sačuva PC+4 u R31, odredište je PC relativno (JAL) ili registarsko (JALR)
TRAP	Transfer kontrole OS-u na vektor adresu
RFE	Return from an exception
Sa pokretnim zarezom	FP operacije nad DP i SP formatima
ADDD, ADDF	Add DP, SP brojeve
SUBD, SUBF	Subtract DP, SP brojeve
MULTD, MULTF	Multiply DP, SP floating point
DIVD, DIVF	Divide DP, SP floating point
CVTF2D, CVTF2I, CVTD2F, CVTD2I, CVTI2F, CVTI2D	Convert instructions: CVTx2y pretvara iz tipa x u tip y, gdje x i y su I (integer), D (double precision), ili F (single precision). Oba operanda su u FPR-ima.
__D, __F	DP i SP poređenja: "__" može biti LT, GT, LE, GE, EQ, NE; postavlja bit u FP status registru

Tabela 4.4. Spisak instrukcija ogledne arhitekture. SP=jednostruka preciznost; DP=dupla preciznost.

Tabela 4.5. daje “rang listu popularnosti” (učestanost korištenja) instrukcija za SPECint92 benchmark, a tabela 4.6. za SPECfp92.

Instrukcija	compress	eqntott	espresso	gcc(ccl)	li	Cjelobrojno prosjek
load	19.8%	30.6%	20.9%	22.8%	31.3%	26%
store	5.6%	0.6%	5.1%	14.3%	16.7%	9%
add	14.4%	8.5%	23.8%	14.6%	11.1%	14%
sub	1.8%	0.3%		0.5%		0%
mul				0.1%		0%
div						0%
compare	15.4%	26.5%	8.3%	12.4%	5.4%	14%
load imm	8.1%	1.5%	1.3%	6.8%	2.4%	4%
cond branch	17.4%	24.0	15.0%	11.5%	14.6	17%
jump	1.5%	0.9%	0.5%	1.3%	1.8%	1%
call	0.1%	0.5%	0.4%	1.1%	3.1%	1%
return, jump ind	0.1%	0.5%	0.5%	1.5%	3.5%	1%
shift	6.5%	0.3%	7.0%	6.2%	0.7%	4%
and	2.1%	0.1%	9.4%	1.6%	2.1%	3%
or	6.0%	5.5%	4.8%	4.2%	6.2%	5%
other (xor, not)	1.0%		2.0%	0.5%	0.1%	1%
load FP						0%
store FP						0%
add FP						0%
sub FP						0%
mul FP						0%
div FP						0%
compare FP						0%
mov reg-reg FP						0%
other FP						0%

Tabela 4.5. Raspodjela instrukcija pri izvršavanju pet SPECint92 programa.

Instrukcija	dodue	car	hydro2d	mdljdp2	su2cor	FP prosjek
load	1.1%	0.2%	0.1%	1.1%	3.6%	1%
store	1.3%	0.1%		0.1%	1.3%	1%
add	13.6%	13.6%	10.9%	4.7%	9.7%	11%
sub	0.3%		0.2%		0.7%	0%
mul						0%
div						0%
compare	3.2%	3.1%	1.2%	0.3%	1.3%	2%
load imm	2.2%		0.2%	2.2%	0.9%	1%
cond branch	8.0%	10.1%	11.7%	9.3%	2.6%	8%
jump	0.9%	0.4%		0.4%	0.1%	0%
call	0.5%	1.9%			0.3%	1%
return, jmp ind	0.6%	1.9%			0.3%	1%
shift	2.0%	0.2%	2.4%	1.3%	2.3%	2%
and	0.4%	0.1%			0.3%	0%
or		0.2%	0.1%	0.1%	0.1%	0%
other (xor, not)						0%
load FP	23.3%	19.8%	24.1%	25.9%	21.6%	23%
store FP	5.7%	11.4%	9.9%	10.0%	9.8%	9%
add FP	8.8%	7.3%	3.6%	8.5%	12.4%	8%
sub FP	3.8%	3.2%	7.9%	10.4%	5.9%	6%
mul FP	12.0%	9.6%	9.4%	13.9%	21.6%	13%
div FP	2.3%		1.6%	0.9%	0.7%	1%
compare FP	4.2%	6.4%	10.4%	9.3%	0.8%	6%
move reg-reg FP	2.1%	1.8%	5.2%	0.9%	1.9%	2%
other FP	2.4%	8.4%	0.2%	0.2%	1.2%	2%

Tabela 4.6. Raspodjela instrukcija pri izvršavanju pet SPECfp92 programa.

4.2. Efektivnost ogledne arhitekture

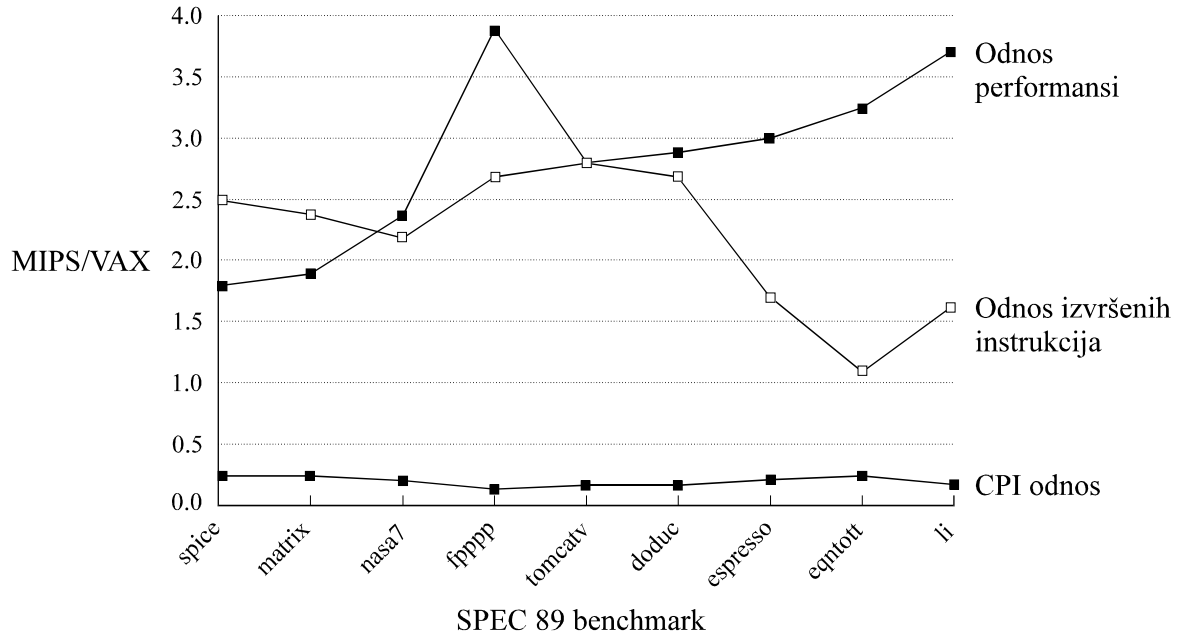
Kao što je rečeno u prethodnim poglavljima, a naročito u 2.5., pred projektanta novog procesora se postavlja jedno važno pitanje - šta je efektivnije, više jednostavnih ili manje složenih instrukcija?”

Brzina izvršenja svakog programa direktno zavisi od broja instrukcija u njemu, prosječnog potrebnog broja ciklusa sata po instrukciji (CPI) i frekvencije sinhronizacionog signala sata. Preciznije, vrijeme izvršenja programa se dobije kada se broj instrukcija pomnoži sa CPI, a zatim se pomnoži sa periodom signala sata.

Dizajneri VAX 8700 su, krajem 1980-tih godina, izveli njegovo kvantitativno poređenje sa MIPS 2000 (slična oglednoj arhitekturi). Slika 4.2. daje odnos brojeva izvršenih instrukcija, odnos CPI-a i odnos performansi mjeren ciklusima sata (uz pretpostavku da su ciklusi sata istog trajanja).

1. MIPS izvršava 2 puta više instrukcija od VAX-a.
2. CPI VAX-a je oko 6 puta veći nego kod MIPS-a.

Zaključak je da MIPS ima skoro 3 puta više performanse i jednostavniji hardver. Zato je proizvođač VAX-a svojevremeno odbacio ovu arhitekturu i prešao na MIPS a kasnije i ALPHA arhitekture.



Slika 4.2. Odnos broja izvršenih instrukcija i performansi po ciklusu sata u SPEC89 programima za MIPS M2000 i VAX 8700.

1970-tih godina, u nastojanju da se eliminiše "semantički procjep" (semantikos - grč. koji označava; semantika - nauka o značenju/filozofija jezika), pravljeni su instrukcijski skupovi "visokog nivoa". Nastojanje da se "pojačavanjem" IS-a podigne HW na nivo jezika zapalo je u ćorsokak. Dajući previše semantičkog sadržaja instrukciji dizajner čini njenu upotrebu mogućom samo u ograničenim situacijama.

1. Ne postoji tipičan program, pa ni procedura za izgradnju optimalnog IS-a.
2. 80x86 arhitektura je primjer kako loša arhitektura može postati uspješna (segmentacija memorije umjesto strananja, prošireni akumulator za cjelobrojne podatke umjesto GPR-a, stack za FP podatke ...).
3. ne postoji arhitektura bez mane (svaka je rezultat niza kompromisa). Ono što je bilo logično prije 10 godina, sada više nije zbog stanja aktuelnih tehnologija.

1950-tih godina arhitekte računarskih sistema su se pretežno bavile ubrzavanjem aritmetičkih operacija.

1960-tih su bile popularne stack-arhitekture zbog podrške tadašnjim HLL-ima i kompajlerima. Brzini rada je doprinijela brzina pristupa registrima (interni stack) a ne način njihovog korištenja.

1970-tih, da bi se smanjila cijena razvoja SW-a, koja je rasla brže od cijene HW-a, pokušana je zamjena SW-a HW-om kroz skup instrukcija visokog semantičkog nivoa. Dizajn ISA je bio u centru pažnje arhitekata.

1980-tih moderni kompajleri i zahtjevi za visokim performansama doveli su do povratka na jednostavnije ISA-e na bazi load/store tipa arhitektura.

Danas je jasno da preovladavaju 32 i 64-bitni adresni IS-i. Projektovanje ISA od početka se vrlo rijetko radi a emulacija ogromne baze programa za 80x86 i binarna kompatibilnost su mnogo aktuelnija pitanja.

4.3. Protočne strukture

Protočna struktura (PS i eng. pipeline) je metoda realizacije CPU-a kod koje se izvršavanje više instrukcija međusobno vremenski preklapa. Pogodno je poređenje sa proizvodnom linijom kod serijske proizvodnje u automobilskoj industriji, gdje postoji mnogo (među)koraka od kojih se svaki obavlja paralelno sa svim ostalim. Svaki korak se zove stanje ili segment protočne strukture. Propusnost PS-e je određena brojem instrukcija/automobila koji iz nje izlaze u jedinici vremena. Vrijeme zadržavanja u svakom segmentu se zove mašinski ciklus i jednako je vremenu zadrške najsporijeg segmenta (obično pretstavlja i clock-ciklus). Ako je trajanje operacije u svakom segmentu idealno uravnoteženo, tada je vrijeme izvršenja jedne instrukcije u idealnim uslovima

$$\frac{\text{vrijeme izvršenja bez PS}-e}{\text{broj segmenata PS}-e}$$

U realnosti je ravnotežu teško postići a uvode se i dodatni poslovi neophodni za funkcionisanje PS-e. Ipak se ubrzanje postiže smanjenjem CPI-a kao i trajanja jednog ciklusa sata. PS koristi paralelizam u izvršavanju sekvenci instrukcija i nevidljiva je za programera.

4.4. Ogledna arhitektura bez protočne strukture

Izvršenje instrukcije podijelimo na do 5 ciklusa (CPI-a) i to:

1. IF - dobavljanje instrukcije
   IR ← Mem[PC]
   NPC ← PC + 4 ; Next PC - privremeni registar
2. ID - dekodiranje instrukcija i dobavljanje operanada iz registara
   A ← Regs[$IR_{6...10}$] ; A - privremeni registar
   B ← Regs[$IR_{11...15}$] ; B - privremeni registar
   Imm ← ((IR_{16})^{16} ## $IR_{16...31}$) ; Imm - privremeni registar za neposredne ; vrijednosti, ovdje se izvodi proširenje ; predznakom do 32 bita

Čitanje registara paralelno je moguće zbog fiksnih lokacija polja u instrukciji. Mogu se čitati registri koji nisu potrebni (ali i ne smetaju).

3. EX - izvršenje i računanje efektivne adrese

U ovom ciklusu slijede jedna od četiri ALU-operacije nad ranije pripremljenim podacima.

3.1. Pristup memoriji (računanje efektivne adrese)
   ALUoutput ← A + Imm ; ALUoutput - privremeni registar

3.2. Registarsko-Registarska ALU instrukcija
   ALUoutput ← A func B

3.3. Registarsko-neposredna ALU instrukcija

ALUoutput ← A op Imm

3.4. Grananje

ALUoutput ← NPC + Imm
Cond ← (A op O)

ALU računa adresu grananja, "op" je relacioni operator (=, ≠) nad A pročitanim u prethodnom ciklusu. Pošto se radi o Load/Store arhitekturi, računanje efektivne adrese i izvršavanje instrukcije se može (kombinovano) smjestiti u jedan ciklus, jer nema potrebe za složenim operacijama.

4. MEM - Pristup memoriji/završetak grananja

U ovom ciklusu su aktivne samo Load, Store i instrukcije grananja.

4.1 Pristup memoriji

```
LDM ← Mem[ALUoutput]        ; Load Data from Memory -
                            ; privremeni registar
Mem[ALUoutput] ← B          ; Store
```

4.2. Grananje

```
if (cond)
    then PC ← ALUoutput
    else PC ← NPC
```

5. WB - Write Back cycle

5.1. Registarsko-registarska ALU instrukcija

$Regs[IR_{16...20}] \leftarrow ALUoutput$

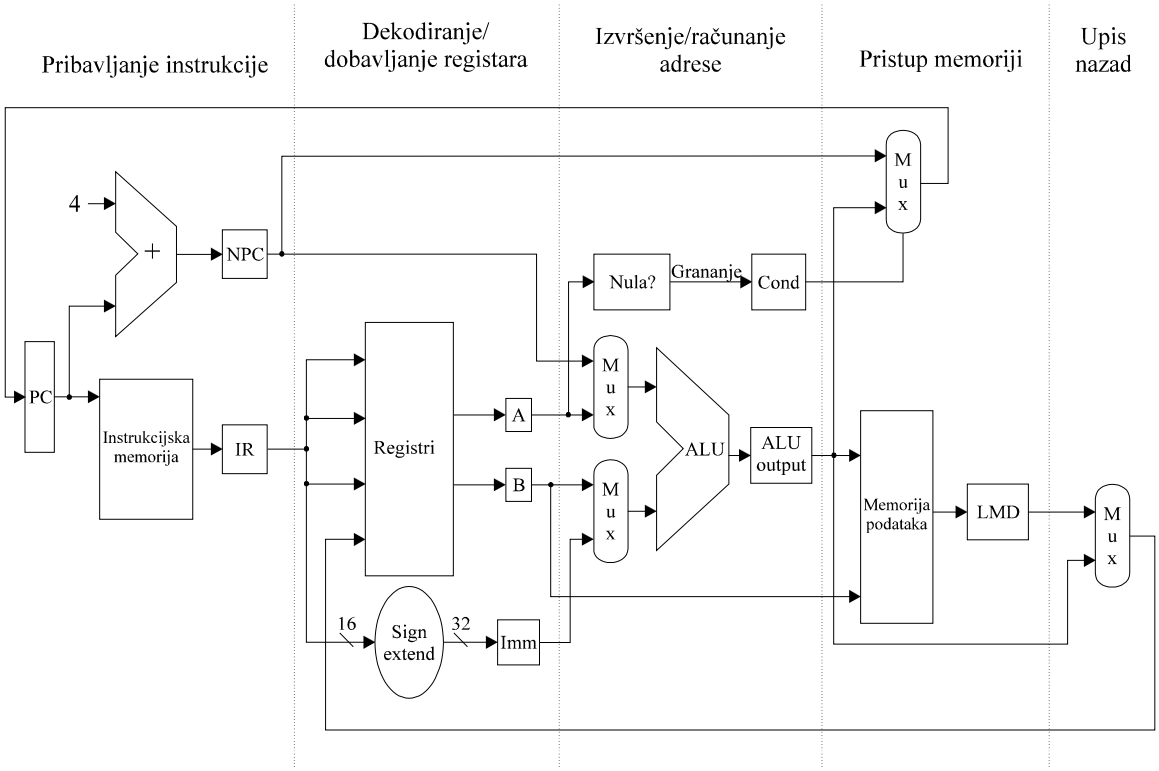
5.2. Registarsko-neposredna ALU instrukcija

$Regs[IR_{11...15}] \leftarrow ALUoutput$

5.3. Load instrukcija

$Regs[IR_{11...15}] \leftarrow LDM$

Ove instrukcije pišu rezultat u registre bilo iz LDM-a ili ALUoutput-a. Odredišni registar je dat u jednom od dva polja i zavisi od OPCODE-a (slika 4.3.).



Slika 4.3. Realizacija puta podataka omogućava da se svaka instrukcija izvrši za četiri ili pet ciklusa. Iako je PC u dijelu za pribavljanje, a registri u dijelu za dekodiranje instrukcija i pribavljanje operanada iz registara, iz obje ove funkcionalne jedinice instrukcija može čitati i u njih pisati.

Na kraju svakog ciklusa njegovi rezultati se upisuju u memoriju ili registre opšte namjene, PC ili privremene registre (LDM, Imm, A, B, NPC, ALUoutput ili Cond).

Instrukcije grananja traju 4, a sve ostale 5 ciklusa. Ovo nije optimizirano jer bi se CPI mogao smanjiti završavanjem ALU instrukcija u MEM ciklusu, pošto su inače u njemu neaktivne.

Upravljačka jedinica za ovakav put podataka bi bila relativno jednostavna sekvencijalna struktura, a za složenije strukture bi se mogao koristiti mikrokod (mikroprogramiranje).

U ovoj neprotočnoj/višeciklusnoj izvedbi dvije ALU bi mogle biti zamijenjene jednom (sa dodatnim MUX-erima) jer nisu obje zauzete u istom ciklusu, ali bi to otežalo uvođenje PS-e.

4.5. Osnovna protočna struktura za arhitekturu oglednog procesora

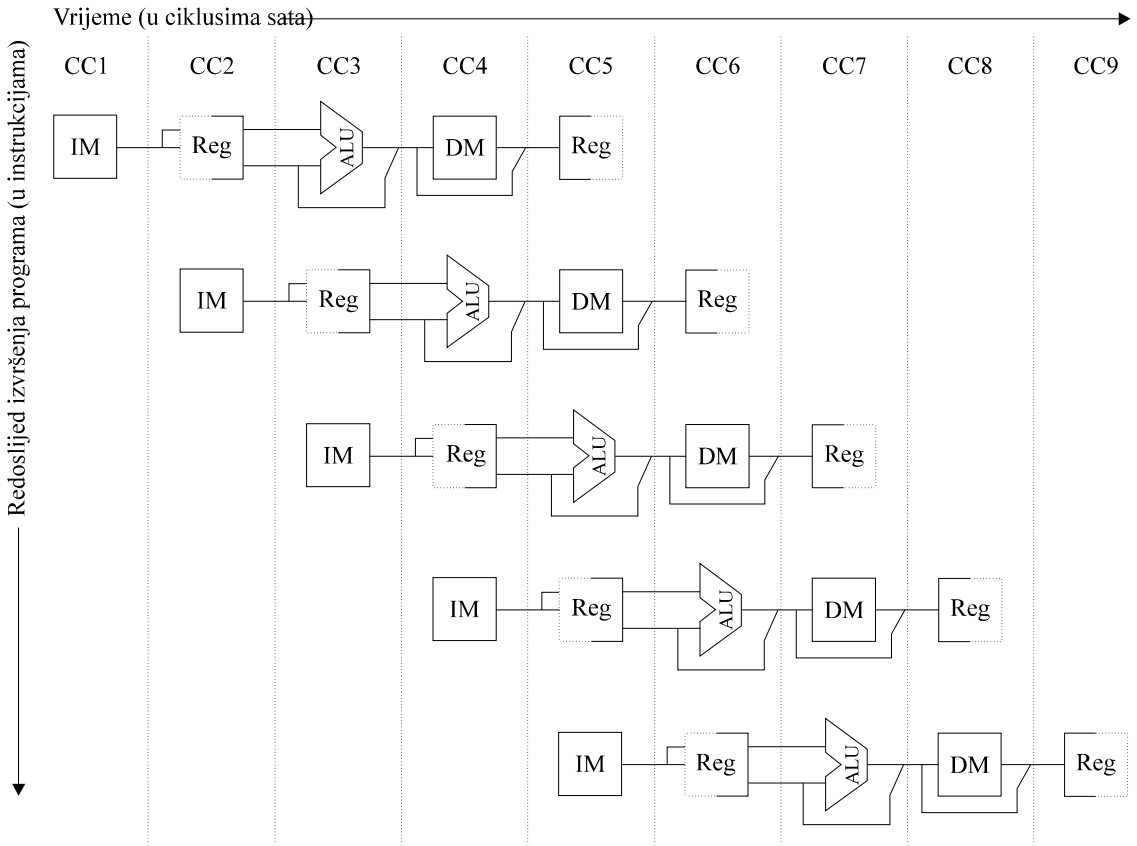
Gotovo bez ikakvih promjena ogledna arhitektura bi mogao počinjati izvršavanje instrukcije svakog ciklusa sata. Tako svaki ciklus prerasta u segment PS-e (tabela 4.7.).

	Broj ciklusa sata								
Broj instrukcije	1	2	3	4	5	6	7	8	9
Instrukcija i	IF	ID	EX	MEM	WB				
Instrukcija i+1		IF	ID	EX	MEM	WB			
Instrukcija i+2			IF	ID	EX	MEM	WB		
Instrukcija i+3				IF	ID	EX	MEM	WB	
Instrukcija i+4					IF	ID	EX	MEM	WB

Tabela 4.7. Jednostavna protočna struktura. Svakog ciklusa sata jedna instrukcija se dobavlja i počinje 5-ciklusno izvršenje. U tom slučaju performanse bi bile pet puta više od istog puta podataka bez protočne

strukture. Imena segmenata su ista kao na slici 3.1. IF=pribavljanje instrukcije, ID=dekodiranje instrukcije, EX=izvršenje, MEM=prostup memoriji i WB=upis nazad (u registre).

Put podataka pretstavljen u duhu PS-e je dat na slici 4.4. Treba paziti da se jedan resurs ne želi koristiti za dvije različite operacije u istom ciklusu sata. U slučaju ogledne arhitekture to je relativno lako, ali inače zahtjeva detaljnu analizu.



Slika 4.4. Protočna struktura se može posmatrati kao niz puteva podataka koji prikazuju stanja izvršenja instrukcija u susjednim ciklusima sata. Prikazano je preklapanje među dijelovima puta podataka, naročito u ciklusu 5 (CC5). Kako se registri koriste kao izvorište u ID-fazi i odredište u WB-fazi, koriste se dva puta u istom ciklusu. Čitanje je označeno punom linijom na lijevoj a pisanje na desnoj strani. IM=memorija instrukcija, DM=memorija podataka, CC=ciklus sata.

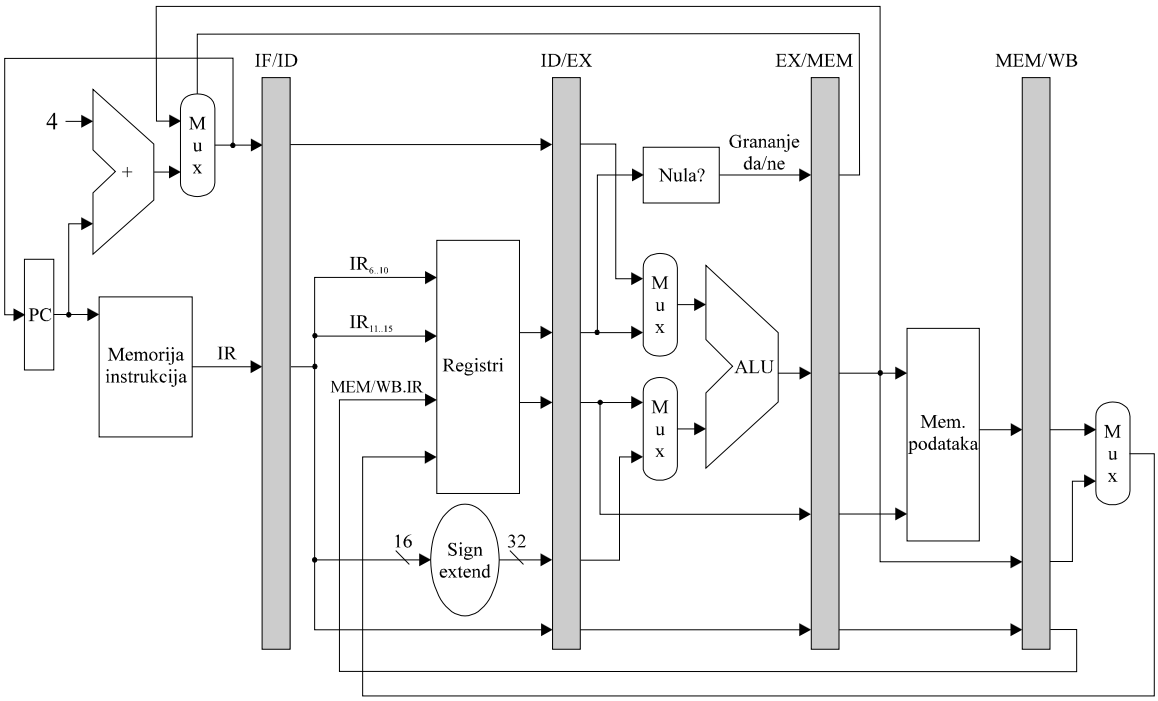
Ogledna arhitektura koristi odvojene memorije za instrukcije i podatke (u realnim procesorima to su odvojeni CACHE-evi), što bi, da nije tako, dovelo do konflikata pri dobavljanju instrukcija i pristupa podacima. Za isti ciklus sata ove memorije moraju imati pet puta veću propusnost nego bez ovakve PS-e.

Registri se koriste u dva segmenta/stanja PS-e, za čitanje u ID i pisanje u WB. Po tome bi bilo potrebno obaviti dva čitanja i jedno pisanje u istom ciklusu (šta ako se traži čitanje i pisanje u isti registar?!). O ovom problemu će biti više riječi kasnije.

Slika 4.4. ne pokazuje ulogu PC-a. Da bi uzimao novu instrukciju svakog clock-a, mora se vršiti inkrementacija i ponovno smještanje u PC svakog ciklusa sata i to u IF segmentu. Tu se javlja problem uslovnih/bezuslovnih grananja koji mijenjaju PC ali ne prije MEM faze PS-e. I o ovom problemu će biti više riječi kasnije.

Svi registri su aktivni u svakom ciklusu sata, sve se u svakom mora završiti za jedan ciklus

sata i to u svim kombinacijama operacija. Pravljenje PS-e zahtijeva da se podaci/vrijednosti proslijeđuju posredstvom veznih registara ili latch-eva. Slika 4.5. prikazuje arhitekturu oglednog procesora sa takvim veznim registrima nazvanim po imenima segmenata koje povezuju. Ovi registri proslijeđuju i podatke i kontrolne signale od jednog do drugog segmenta (polje registra u koji treba upisati rezultat Load ili ALU instrukcije koji daje MEM/WB a ne IF/ID trenutne instrukcije).



Slika 4.5. Put podataka je pretvoren u protočnu strukturu dodavanjem pregradnih-veznih registara između segmenata protočne strukture. Oni prenose vrijednosti i upravljačke informacije od jednog do drugog segmenta.

Svaka instrukcija je aktivna u samo jednom segmentu u jednom trenutku (između dva vezna registra). Tabela 4.8. pokazuje šta se mora dešavati u pojedinim segmentima zavisno od tipa instrukcije koja se izvršava. Polja u registrima su nazvana po podacima koje proslijeđuju.

Segm.	**Bilo koja instrukcija**		
IF	IF/ID.IR←Mem[PC]; IF/ID.NPC, PC←(if EX/MEM.cond {EX/MEM. ALUOutput } else {PC+4});		
ID	ID/EX.A←Regs[IF/ID.$IR_{6..10}$]; ID/EX.B←Regs[IF/ID.$IR_{11..15}$]; ID/EX.NPC←IF/ID.NPC; ID/EX.IR←IF/ID.IR; ID/EX.Imm←$(IF/ID.IR_{16})^{16}$##IF/ID.$IR_{16..31}$;		
	ALU instrukcija	**Čitanje ili pisanje u mem.**	**Instrukcija grananja**
EX	EX/MEM.IR←ID/EX.IR;	EX/MEM.IR←ID/EX.IR	
	EX/MEM.ALUOutput← ID/EX.A *func* ID/EX.B; *or* EX/MEM.ALUOutput← ID.EX.A *op* ID/EX.Imm;	EX/MEM.ALUOutput← ID/EX.A+ID/EX.Imm;	EX/MEM.ALUOutput← ID/EX.NPC+ID/EX.Imm;
	EX/MEM.cond←0;	EX/MEM.cond←0;	EX/MEM.cond← (ID/EX.A *op* 0);
		EX/MEM.B←ID/EX.B;	

MEM	MEM/WB.IR←EX/MEM.IR;	MEM/WB.IR←EX/MEM.IR;
	MEM/WB.ALUOutput← EX/MEM.ALUOutput;	MEM/WB.LMD← Mem[EX/MEM.ALUOutput]; *or* Mem[EX/MEM.ALUOutput]← EX/MEM.B;
WB	Regs[MEM/WB.$IR_{16..20}$]← MEM/WB.ALUOutput; *ili* Regs[MEM/WB.$IR_{11..15}$]← MEM/WB.ALUOutput;	Regs[MEM/WB.$IR_{11..15}$]← MEM/WB.LMD;

Tabela 4.8. Događaji u svakom segmentu protočne strukture. U IF-segmentu, pored dobavljanja instrukcija i računanja nove vrijednosti PC-a, smješta se inkrementirani PC i u PC i u pregradni registar (NPC) da bi se kasnije koristio za računanje odredišne adrese grananja. U ID-fazi dobavljaju se operandi iz registara, znakom proširuje donjih 16 bita IR-a i prosljeđuje IR i NPC. Za vrijeme EX izvodi se ALU operacija ili računanje adrese. Prosljeđuje se IR i B-registar (ako se radi o instrukciji pisanja u memoriju) i postavlja cond u 1 ako je ispunjen uslov grananja. U MEM fazi se odrađuje ciklus pristupa memoriji, odlučuje o grananju, upisuje PC, ako je potrebno, i prosljeđuju vrijednosti potrebne u zadnjem segmentu. U WB se vrši upis u registre izlaza iz ALU-a ili pročitanog iz memorije.

Za upravljanje PS-om treba definisati upravljanje nad 4 MUX-a iz slike 3.4. Dva MUX-a ispred ALU-e rade u zavisnosti od tipa instrukcija (IR-polje u ID/EX registru). Gornji zavisi od toga da li je instrukcija grananja ili ne, a donji zavisi od toga da li je instrukcija registarsko-registarska ALU ili neka druga. MUX u IF segmentu bira između trenutnog PC-a i vrijednosti EX/MEM.NPC-a (adresa grananja) kao adrese sljedeće instrukcije. Njega kontroliše polje EX/MEM.cond. Četvrti MUX je određen time da li je instrukcija u WB segmentu load ili ALU-instrukcija.

Postoji još jedan MUX koji nije na slici 4.5. ali je njegovo postojanje jasno iz WB segmenta ALU-instrukcije. Polje odredišnog registra je na jednom od dva mjesta, zavisno od tipa instrukcije (reg/reg ALU ili bilo ALU-neposredno ili Load). MUX je neophodan za izbor pravog polja IR-a u MEM/WB registru za izbor odredišnog polja. Tabela 4.8. prikazuje događaje u protočnoj strukturi po segmentima i za različite tipove instrukcija.

PS povećava broj izvršenih instrukcija u jedinici vremena ali ne ubrzava izvršenje svake pojedine instrukcije. Naprotiv, čak ga i usporava zbog dodatnih poslova u upravljanju PS-om. Dodatno trošenje vremena dolazi i od kašnjenja pri prolasku kroz pregradne registre i neuravnoteženosti potrošnje vremena u pojedinim segmentima.

5. Hazardi - osnovni problemi kod protočnih struktura

Hazardi su glavne prepreke na putu projektovanja i primjene protočih struktura. To su situacije u kojima je onemogućeno izvršavanje slijedeće instrukcije (u njenom ciklusu). Time se ubrzanja postignuta PS-om smanjuju.

Postoje tri vrste hazarda:

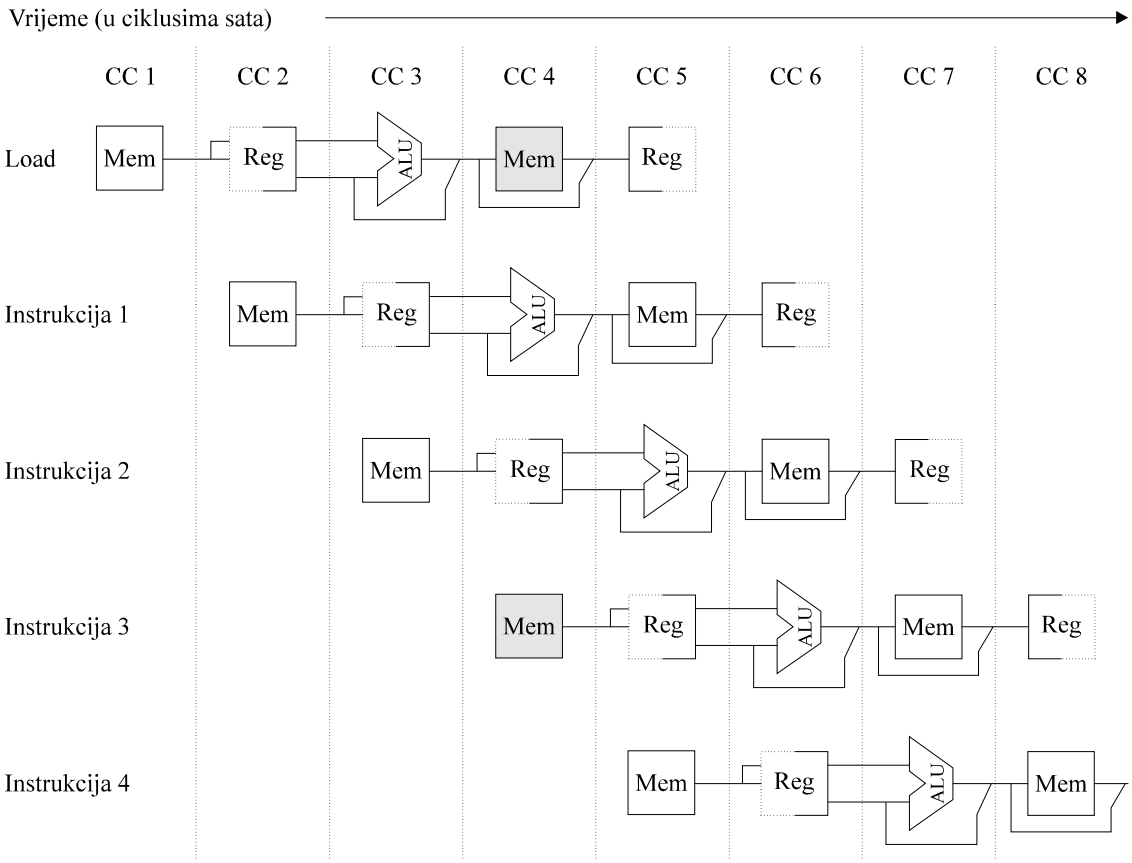
1. **STRUKTURNI hazardi** nastaju kao posljedica nastanka konflikata nad resursima (kada dvije ili više instrukcija zahtijevaju korištenje istog resursa u istom ciklusu).
2. **Hazardi PODATAKA** se javljaju kada jedna instrukcija zavisi od rezultata drugih (jedne ili više) koje još nisu izvršene (još su u PS-i).
3. **UPRAVLJAČKI hazardi** se javljaju kod izvršavanja instrukcija grananja i drugih instrukcija koje mijenjaju sadržaj PC-a.

Hazardi u PS-ama mogu dovesti do zastoja (eng. stall) slično kao kod promašaja pri pristupu kešu. U PS-ama su zastoji složeniji. Eliminisanje hazarda bi zahtijevalo da se neke instrukcije u PS-i (iza hazarda) nastave izvršavati, a neke (prije hazarda) odgode. Za vrijeme zastoja, ne pribavljaju se nove instrukcije (promašaj u kešu izaziva zastoj svih).

5.1. Strukturni hazardi

Strukturni hazardi nastaju u dva slučaja:

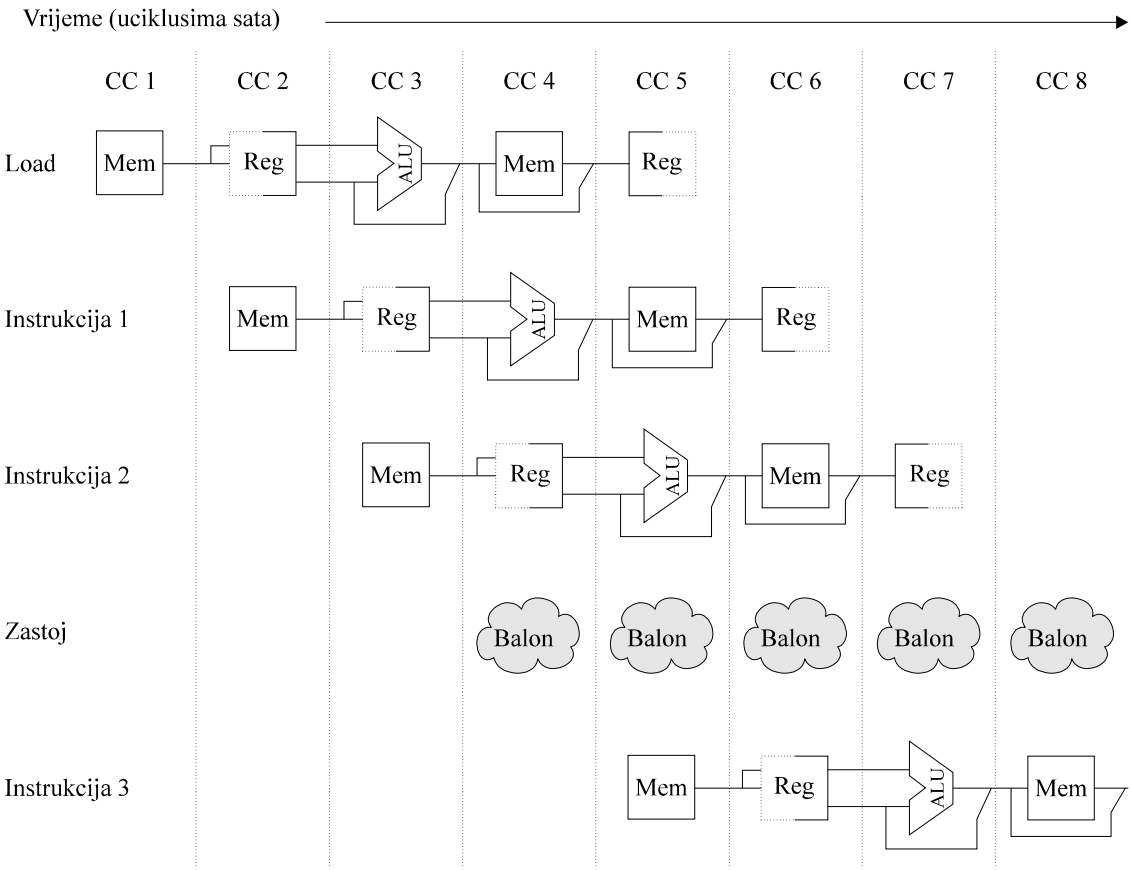
1. ako funkcionalna jedinica nije u potpunosti “protočna” pa se narušava ritam segmenata PS-e,
2. ako nema dovoljno (dupliciranih) resursa za sve kombinacije instrukcija u PS-i (npr. jedan a ne dva porta za pristup registrima). Zastoj rada odgađa jednu instrukciju dok se druga ne usluži - time se povećava CPI na više od 1. Slično bi bilo sa jedinstvenom memorijom za instrukcije i podatke - slika 5.1., a na slici 5.2. je dato rješenje hazarda pomoću zastoja ili "mjehurića" (trošenje vremena bez korisnog rada). Tabela 4.1. to prikazuje kao pomijeranje (odgađanje) i+3 instrukcije udesno jedno mjesto.



Slika 5.1 Arhitektura sa samo jednim memorijskim portom će generisati konflikt uvijek kada instrukcija pristupa memoriji (u ovom slučaju Load).

	Broj ciklusa sata									
Instrukcija	1	2	3	4	5	6	7	8	9	10
Load instrukcija	IF	ID	EX	MEM	WB					
Instrukcija i+1		IF	ID	EX	MEM	WB				
Instrukcija i+2			IF	ID	EX	MEM	WB			
Instrukcija i+3				zastoj	IF	ID	EX	MEM	WB	
Instrukcija i+4						IF	ID	EX	MEM	WB
Instrukcija i+5							IF	ID	EX	MEM
Instrukcija i+6								IF	ID	EX

Tabela 5.1. Protočna struktura u zastoju - čitanje memorije sa jednim portom. U ciklusu 4 se ne pokreće nijedna instrukcija. Sve prethodne instrukcije mogu nastaviti sa procesiranjem. U ciklusu 8 neće biti završene instrukcije.



Slika 5.2. Strukturni hazard izaziva ubacivanje balona/mjehurića u protočnu strukturu.

Strukturni hazardi se mogu izbjeći dupliranjem resursa. To je ponekad preskupo i zbog dodatnog kašnjenja koje se time izaziva:

1. dual-port memorija za instrukcije i podatke mora imati dvostruko veću propusnost,
2. FP-množač je složena struktura, ne izaziva često zastoje, pa se kao kompromis, ili izbacuje ili se realizuje kao neprotočna ili djelimično protočna struktura (da bi se i smanjila kašnjenja u veznim registrima).

5.2. Hazardi podataka

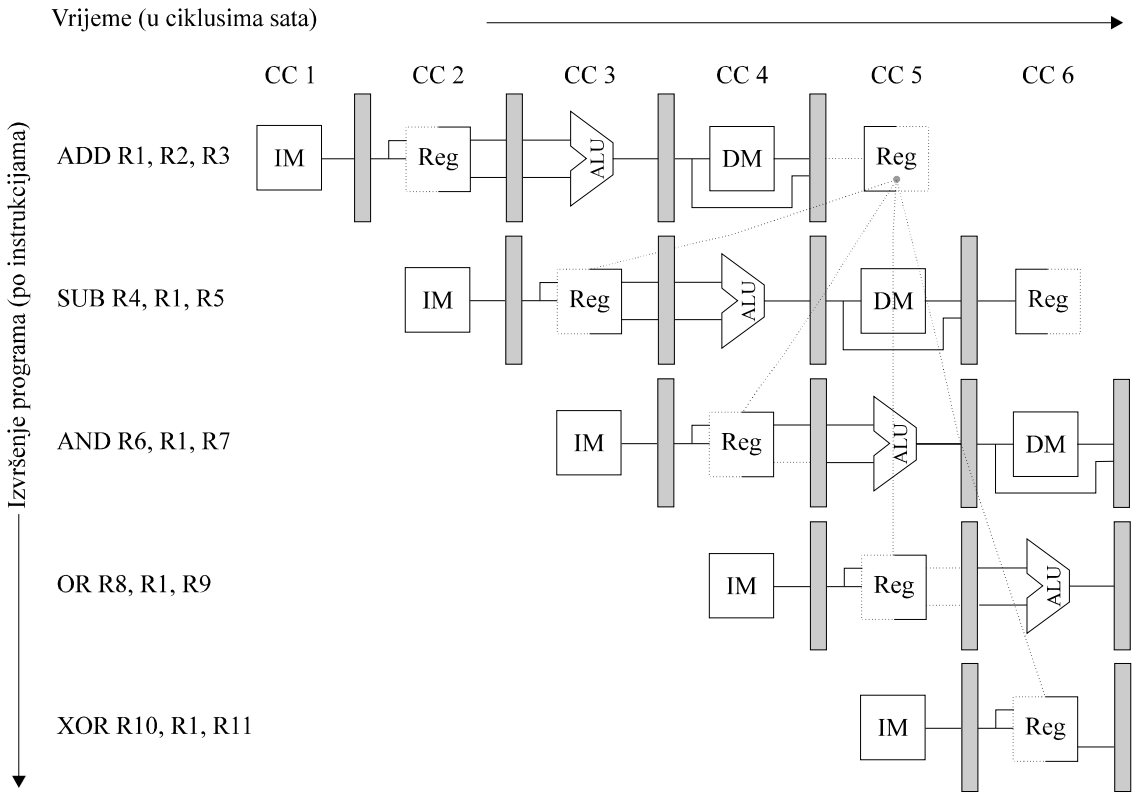
Hazardi podataka nastaju kada paralelno izvršavanje u PS-i promijeni redoslijed čitanja ili pisanja operanada u odnosu na sekvencijalno izvršenje (bez PS-e).

Primjer:

```
ADD   R1, R2, R3
SUB   R4, R5, R1
AND   R6, R1, R7
OR    R8, R1, R9
XOR   R10, R1, R11
```

Iza ADD instrukcije, sve ostale koriste njen rezultat (slika 5.3.). Ako se ništa ne promijeni, SUB će čitati pogrešne podatke. Isto bi se desilo i sa AND instrukcijom jer će rezultat u R1 biti upisan tek u CC5, a SUB ga treba čitati u CC3 i AND u CC4. Ako su registri pravilno dijeljeni (u dva podciklusa - u prvom pisanje a u drugom čitanje), OR će se korektno obaviti.

XOR instrukcija bi se svakako korektno obavila - bez hazarda podataka.



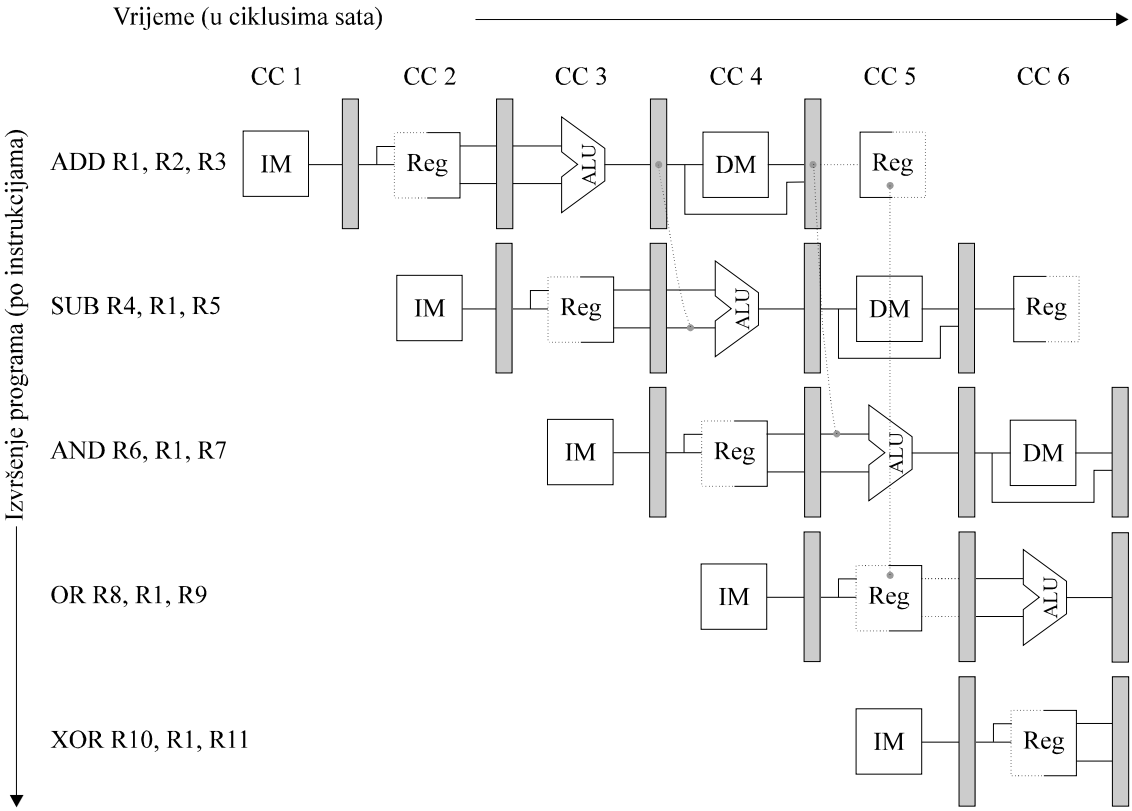
Slika 5.3. Korištenje rezultata ADD instrukcije u u naredne tri instrukcije izaziva hazard jer njen rezultat nije upisan u registar prije nego što ga naredne tri čitaju.

Riješenje hazarda podataka je:

1. zastojima - mjehurićima (skupo) ili prekidima (isti efekat),
2. proslijeđivanjem podataka unaprijed (eng. forwarding, bypassing, short-circuiting).

Proslijeđivanje je moguće jer SUB ne treba rezultat u R1 prije nego ga ADD izračuna. Zato je proslijeđivanje moguće ako se rezultat ALU-a iz EX/MEM registra vraća uvijek na ALU-ulazne latch-eve (kroz MUX) pa ako upravljačka struktura otkrije da je prethodna ALU-operacija upisivala rezultat u registar koji je izvorišni za trenutnu ALU-operaciju, proslijeđuje (izabira proslijeđene) rezultate kao ulaz u ALU umjesto zastarjelih vrijednosti pročitanih iz registara.

Nekada je potrebno proslijeđivanje rezultata instrukcije koja je počela tri ciklusa ranije. Slika 5.4. pokazuje izvršenje sekvence instrukcija sa proslijeđivanjem i bez zastoja.



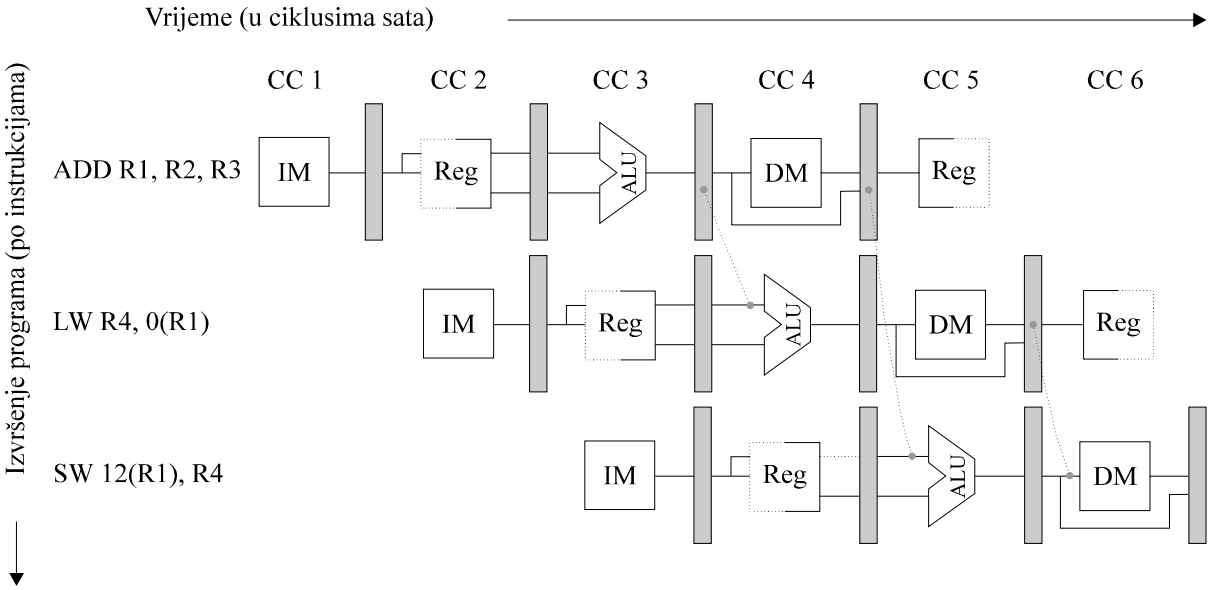
Slika 5.4. Skup instrukcija koje zavise od rezultata ADD koriste puteve prosljeđivanja da bi izbjegle hazarde. Ulaz podataka za SUB i AND instrukcije se prosljeđuje iz EX/MEM i MEM/WB pregradnjih registara, respektivno, na prvi ulaz ALU. OR dobija svoje podatke kroz skup registara, što se postiže čitanjem registara u drugoj polovini, a pisanjem u prvoj polovini ciklusa (kao što je prikazano isprekidanim linijama). Proslijeđeni podaci mogu doći na bilo koji od ALU ulaza.

U opštem slučaju, prosljeđivanje je moguće direktno sa izlaza jedne funkcionalne jedinice na ulaz one kojoj je potreban.

Primjer:

```
ADD   R1, R2, R3
LW    R4, 0(R1)
SW    12(R1), R4
```

Da bi se izbjegao zastoj treba proslijediti R1 i R4 iz veznih registara do ALU-a i ulaza u memoriju podataka. Slika 5.5 pokazuje puteve prosljeđivanja. Može se tražiti prosljeđivanje iz bilo kojeg veznog registra do ulaza bilo koje funkcionalne jedinice.



Slika 5.5. Upis u memoriju zahtijeva operand za vrijeme MEM ciklusa, a njegovo prosljeđivanje je ovdje prikazano. Rezultat čitanja iz memorije se prosljeđuje sa izlaza memorije u MEM/WB na ulaz mamorije radi upisa.Pored toga, izlaz ALU se prosljeđuje na njen ulaz radi računanja adresa za pristup memoriji.

5.2.1. Klasifikacija hazarda podataka

Uzimajući u obzir samo hazarde u kojima se koriste registri CPU-a, u zavisnosti od redoslijeda čitanja i pisanja u instrukcijama "i" (ranija) i "j" (kasnija) postoje slijedeći mogući hazardi:

1. **RAW** (**Read-After-Write**) - se javlja kada instrukcija "j" pokušava da čita izvor prije nego što "i" upiše rezultat u njega. Ovo je najčešći hazard i opisan je u slikama 5.4. i 5.5.
2. **WAW** (**Write-After-Write**) - se javlja kada "j" pokušava pisati prije nego to "i" uradi. Pogrešan redoslijed ostavlja pogrešan operand upisan u odredište. Ovo je moguće u PS-ama sa upisom u više od jednog segmenta (ili kada se instrukcija nastavlja izvršavati iako je prethodna u zastoju - ogledna cjelobrojna PS ovo izbjegava). Tada je moguće i da dvije instrukcije pokušaju pisati u istom ciklusu sata.
3. **WAR** (**Write-After-Read**) - se javlja kada "j" pokuša pisati po odredištu prije nego "i" pročita njegov sadržaj, pa "i" pročita pogrešan sadržaj. Ovo je izbjegnuto u oglednoj arhitekturi ranim čitanjem (ID-segment) i kasnim pisanjem (WB). Javlja se kod PS-a sa instrukcijama sa ranjim pisanjem i kasnim čitanjem. Ovo nije uobičajeno pa je WAR-hazard rijetkost.
4. **RAR** (**Read-After-Read**) ne pretstavlja hazard.

5.3. Hazardi podataka koji zahtjevaju zastoje

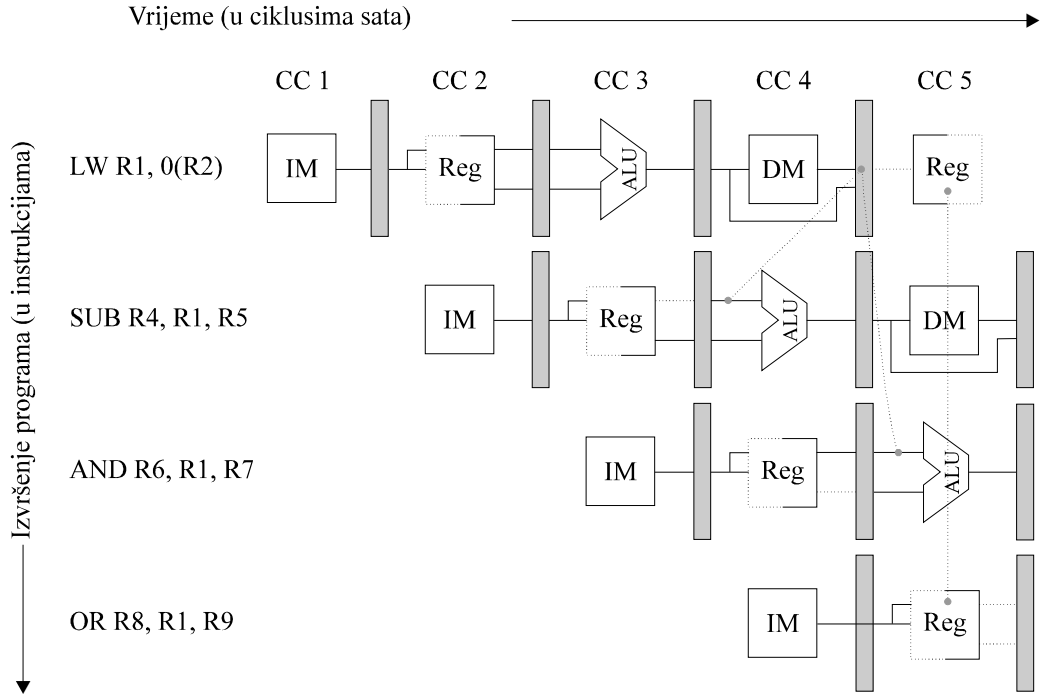
Ne mogu se svi hazardi podataka riješiti prosljeđivanjem.

Primjer:

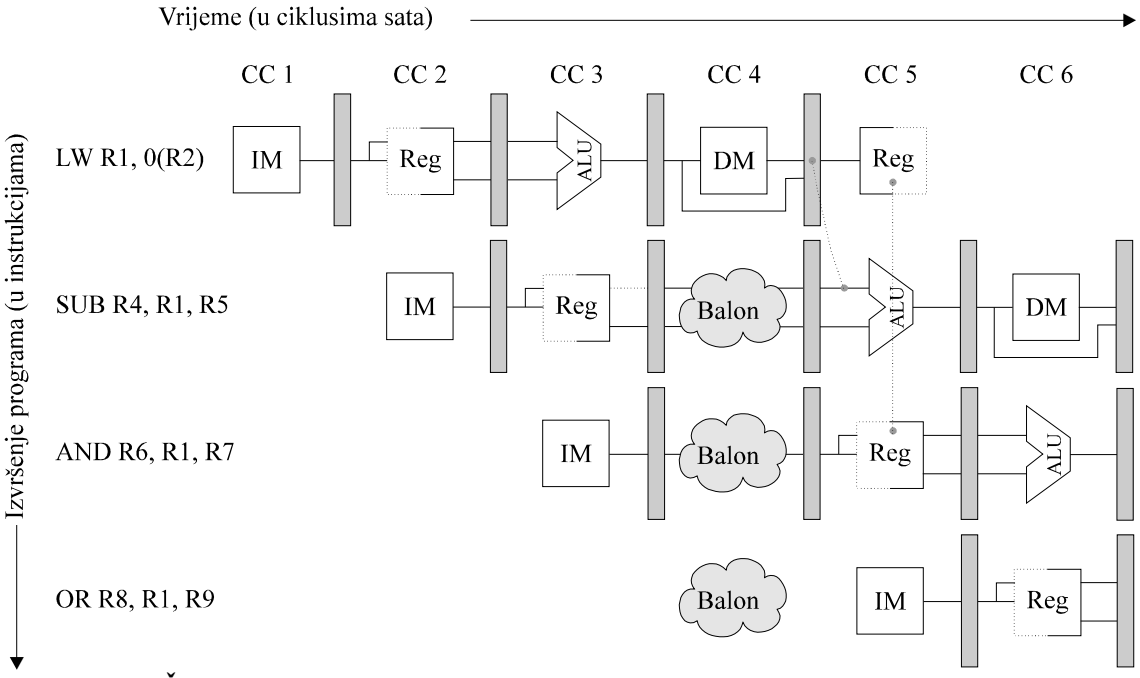
```
LW    R1, 0(R2)
SUB   R4, R1, R5
```

```
AND   R6, R1, R7
OR    R8, R1, R9
```

Slika 5.6. prikazuje potrebna prosljeđivanja. LW ima podatak na kraju CC4 dok SUB traži podatak na početku istog ciklusa (vremenski unazad - nemoguće proslijediti). Može se proslijediti u ALU iz MEM/WB registra za AND instrukciju koja počinje dva ciklusa iza LW. OR se obavlja bez problema jer dobija traženu vrijednost kroz registre (R1). Kako LW kasni sa podatkom - potreban je dodatni hardver (eng. pipeline interlock) koji otkriva hazard i pravi zastoj u PS-i dok se hazard ne riješi (slično kao kod strukturnih hazarda - mjehurićima). Nakon jednog ciklusa zastoja, prosljeđivanje (slika 5.7.) u AND ide kroz skup registara. Tabela 5.2. daje prikaz protočne strukture prije i poslije uvođenja zastoja.



Slika 5.6. Instrukcija čitanja iz memorije može proslijediti svoje rezultate AND i OR instrukcijama, ali ne i SUB, jer je to vremenski nemoguće.



Slika 5.7. Čitanje iz memorije izaziva zastoj u ciklusu 4, kasneći instrukciju SUBi one iza nje. Ovo kašnjenje omogućava uspješno prosljeđivanje vrijednosti u sljedećem ciklusu sata.

LW R1, 0(R1)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R1)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	zastoj	EX	MEM	WB	
AND R6, R1, R7			IF	zastoj	ID	EX	MEM	WB
OR R8, R1, R9				zastoj	IF	ID	EX	MEM	WB

Tabela 5.2. U gornjem dijelu tabele vidi se zašto je zastoj potreban: MEM ciklus čitanja iz memorije daje vrijednost koja je potrebna u EX fazi SUB instrukcije koja se dešava istovremeno. Problem je riješen ubacivanjem zastoja, kao što je prikazano u donjem dijelu tabele.

5.4. Kompajlersko raspoređivanje instrukcija

Kompajleri pisani za procesore sa protočnim strukturama imaju veliku ulogu u iskorištavanju mogućnosti procesora. Zato pisci kompajlera moraju dobro poznavati arhitekturu procesora, ili vrlo blisko sarađivati sa arhitektama. U protivnom, efikasno izvršavanje kompajliranog koda može biti narušena.

Primjer: iskaz A=B+C se tipično prevodi u sekvencu instrukcija kao u tabeli 5.3. ADD mora sačekati da se učita C, SW ne mora čekati jer rezultat iz ALU-a se može proslijediti na ulaz memorije podataka.

LW R1, B	IF	ID	EX	MEM	WB	
LW R2, C		IF	ID	EX	MEM	WB

ADD R3, R1, R2	IF	ID	zastoj	EX	MEM	WB	
SW A, R3		IF	zastoj	ID	EX	MEM	WB

Tabela 5.3. Sekvenca koda ogledne arhitekture za A=B+C. Instrukcija ADD se mora zadržati da omogući završetak čitanja C. SW ne treba dodatno zadržavati jer je moguće proslijediti rezultat iz MEM/WB direktno na ulaz memorije podataka radi upisa/smještanja.

Izbjegavanje zastoja u ovako kratkoj sekvenci instrukcija je teško postići. Međutim, kod nešto većeg posla i nešto složenije sekvence instrukcija, dobar kompajler može značajno pomoći u izbjegavanju zastoja.

Za slučaj: a = b + c
d = e - f

kompajler bi mogao generisati sekvencu:

```
LW    Rb, b
LW    Rc, c
LW    Re, e          ; zamjena instrukcija za izbjegavanje zastoja
ADD   Ra, Rb, Rc
LW    Rf, f
SW    a, Ra          ;store i load zamjenjeni da se izbjegne zastoj
SUB   Rd, Re, Rf
SW    d, Rd
```

i eliminisati zastoje (uz korištenje mehanizma prosljeđivanja).

5.5. Upravljanje oglednom protočnom strukturom

Svi hazardi podataka se mogu provjeravati u ID fazi protočne strukture. Ako hazard podataka postoji, instrukcije se zaustavljaju prije početka izvršavanja - prelaska iz ID u EX fazu. U istoj fazi se može odrediti koje prosljeđivanje će biti potrebno i postaviti odgovarajuće upravljanje.

Otkrivanje hazarda (interlock-a) u ranoj fazi PS-e pojednostavljuje HW - nikada se ne suspenduje instrukcija koja je ušla u fazu izvršenja (promjene stanja mašine) osim ako se zaustavlja čitava mašina (npr. zbog promašaja u kešu).

Alternativno, hazard se može otkriti na početku faze kada se koristi operand (kod ogledne PS je to EX i MEM faza).

Primjer: RAW sa izvorištem iz LOAD instrukcije (load interlock) se može realizovati provjerom u ID fazi dok se prosljeđivanje do ALU-ulaza može napraviti u EX-fazi. Tabela 5.4. daje različite situacije koje se moraju rješavati. Ako postoji RAW hazard sa LOAD izvorom - LOAD instrukcija će biti u EX fazi kada je instrukcija koja traži taj podatak u ID fazi. Sve kombinacije ovakvih hazarda su date u tabeli 5.5.

Stanje	Primjer sekvence koda	Akcija

Nema zavisnosti	LW **R1**, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Nije moguć hazard jer ne postoji zavisnost od R1 u sljedeće tri instrukcije.

Zavisnost zahtijeva zastoj	LW **R1**, 45(R2)
	ADD R5, **R1**, R7
	SUB R8, R6, R7
	OR R9, R6, R7
		Komparatori otkrivaju upotrbu R1 u ADD izadržavajuADD (i SUB i OR) prije nego što ADD počne EX.

Zavisnost se rješava	LW **R1**, 45(R2)	Komparatori otkrivaju upotrbu R1 u SUB i

prosljeđivanjem	ADD R5, R6, R7
	SUB R8, **R1**, R7
	OR R9, R6, R7
		prosljeđuju rezultat čitanja iz memorije ALU-uu trenutku kada SUB počinje EX.

Zavisnost sa pristupanjem redom	LW **R1**, 45(R2)
	ADD R5, R6, R7
	SUB R8, R6, R7
	OR R9, **R1**, R7
		Nikakva akcija nije potrebna jer čitanje R1 od strane OR se javlja u drugoj polovini ID faze, dok se pisanje pročitanjih podataka dešava u prvoj polovini.

Tabela 5.4. Situacije koje hardver za otkrivanje hazarda može otkriti poređenjem odredišta i izvorišta susjednih instrukcija.

Opkod polje u ID/EX ($ID/EX.IR_{0..5}$)	Opkod polje u IF/ID ($IF/ID.IR_{0..5}$)	Odgovarajuća polja operanada
Load	Register-register ALU	$ID/EX.IR_{11..15}$= $IF/ID.IR_{6..10}$
Load	Register-register ALU	$ID/EX.IR_{11..15}$= $IF/ID.IR_{11..15}$
Load	Load, store, ALU immediate, or branch	$ID/EX.IR_{11..15}$= $IF/ID.IR_{6..10}$

Tabela 5.5. Da bi se otkrila potreba za zastojem nakon čitanja iz memorije, potrebna su tri poređenja. Prva i druga linija u tabeli testiraju da li je odredišni registar kod čitanja memorije, jedan od izvorišnih za registar-u-registar operaciju u ID. Treća linija određuje da li je odredišni registar izvorište adrese za pristup memoriji., neposredna vrijednost za ALU ili test grananja. IF/ID registar drži stanje instrukcije u ID, koja potencijalno koristi rezultat čitanja iz memorije, dok ID/EX sadrži stanje instrukcije u EX, koja je potencijalno instrukcija čitanja iz memorije.

Kada je hazard otkriven, upravljačka struktura mora izazvati zastoj instrukcija iz IF i ID faze, promjenom upravljačkog dijela ID/EX registra u npr. nule (no-op) ili npr.

ADD R0, R0, R0

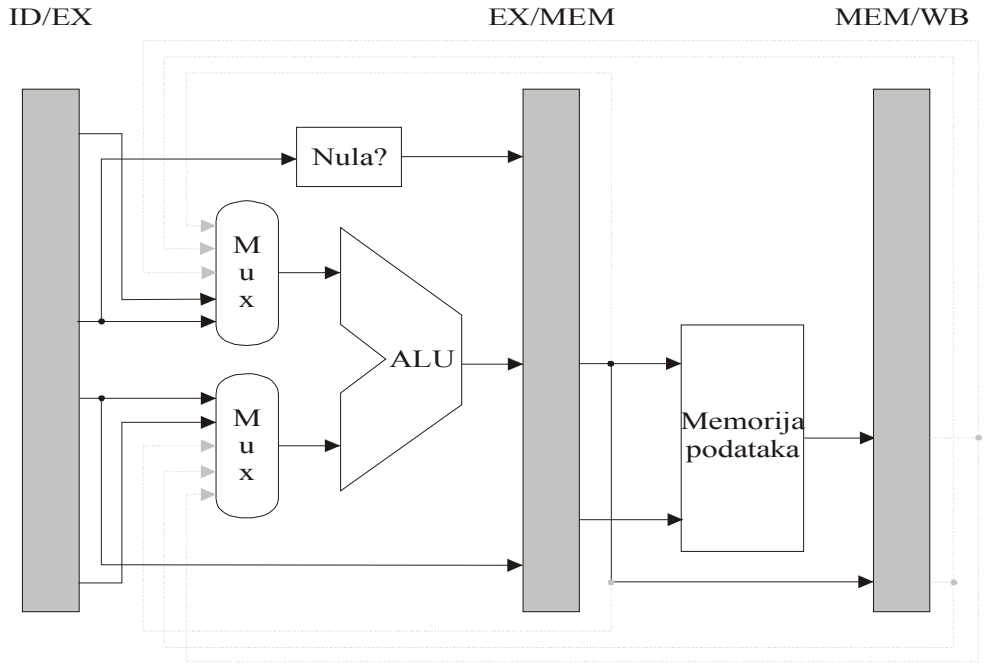
istovremeno recirkulišući sadržaj IF/ID registra da bi se zadržale instrukcije u zastoju. Hazardi se mogu otkriti poređenjem sadržaja pregradnih registara PS-e i ubacivanjem no-op-a.

Pregradni registar koji sadrži izvorišnu instrukciju	Opkod izvorišne instrukcije	Pregradni registar koji sadrži odredišnu instrukciju	Opkod odredišne instrukcije	Odredište proslijeđenog rezultata	Poređenje (ako je jednako tada proslijedi)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{16..20}$ =$ID/EX.IR_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{16..20}$ =$ID/EX.IR_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{16..20}$ =$ID/EX.IR_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{16..20}$ =$ID/EX.IR_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{11..15}$ =$ID/EX.IR_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{11..15}$ =$ID/EX.IR_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{11..15}$ =$ID/EX.IR_{6..10}$

MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.$IR_{11..15}$ =ID/EX.$IR_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.$IR_{11..15}$ =ID/EX.$IR_{6..10}$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.$IR_{11..15}$ =ID/EX.$IR_{11..15}$

Tabela 5.6. Prosljeđivanje podataka ka dva ALU ulaza (za instrukciju u EX) se može desiti izALU rezultata (u EX/MEM ili MEM/WB) ili iz rezultata čitanja memorije u MEM/WB. Postoji 10 poređenje koja su neophodna da bi se ustanovilo da li je neophodno prosljeđivanje.

Upravljanje logikom za prosljeđivanje je slično ali ima više varijacija. Ključno je to što pregradni registri sadrže i podatke koji se prosljeđuju i polja izvorišnih i odredišnih registara. Sva prosljeđivanja se dešavaju iz ALU ili memorije podataka u ALU ulaz, ulaz memorije podataka ili jedinicu za otkrivanje nule. Prosljeđivanje se može realizovati poređenjem odredišnih registara iz IR polja iz EX/MEM i MEM/WB pregradnih registara. Tabela 5.6. daje poređenja i moguća prosljeđivanja gdje je odredište prosljeđivanja ALU-ulaz, za instrukciju koje je trenutno u EX-fazi. Pored komparatora i logike za omogućavanje proslijeđivanja, potrebno je povećati broj ulaza u MUX-e na ulazu u ALU i dodati veze između pregradnih registara koji se koriste za proslijeđivanje (Slika 5.8.).



Slika 5.8. Prosljeđivanje rezultata ALU-u zahtijeva dodatna tri ulaza na svaki ulazni MUX i tri dodatna puta za te ulaze - izlaz ALU-a na kraju EX, izlaz ALU-a na kraju MEM i izlaz iz memorije na kraju MEM.

5.6. Upravljački hazardi

Upravljački hazardi (eng. control hazards) mogu više naškoditi performansama PS-e nego hazardi podataka. Ako je "i" instrukcija grananja (ispunjen uslov), PC se mijenja tek na kraju MEM faze (kada se izračuna adresa i obavi poređenje - testiranje uslova). Najjednostavnije je uvesti zastoj (dva ciklusa) čim se dekodira instrukcija grananja (tabela 5.7.). Nakon zastoja IF

se mora ponoviti (za razliku od običnih hazarda podataka!!!) čim se sazna ishod grananja. Prvi IF je efektivno ništa - zastoj, jer se ne nastavlja. Tri ciklusa gubitka obaraju performanse PS i do 50%.

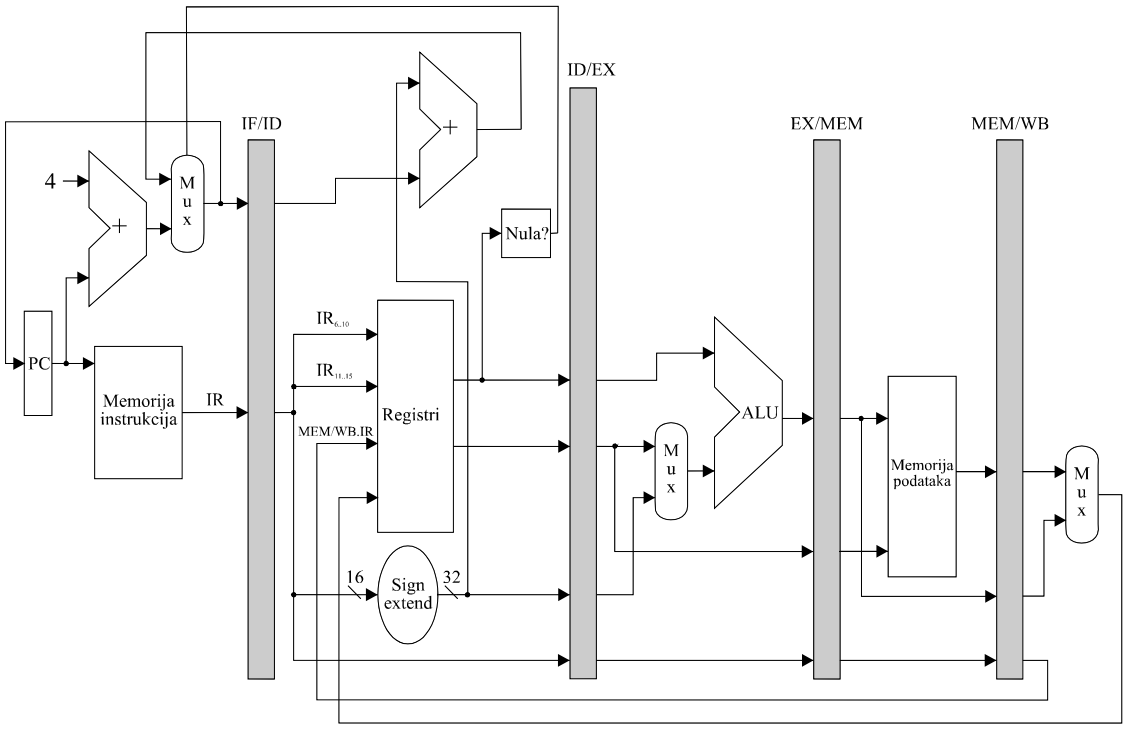
Instrukcija grananja	IF	ID	EX	MEM	WB					
Instr. iza grananja		IF	zastoj	zastoj	ID	EX	MEM	WB		
Instr. iza grananja +1						IF	ID	EX	MEM	WB
Instr. iza grananja +2							IF	ID	EX	MEM
Instr. iza grananja +3								IF	ID	EX
Instr. iza grananja +4									IF	ID
Instr. iza grananja +5										IF

Tabela 5.7. Grananje izaziva zastoj od tri ciklusa u oglednoj protočnoj strukturi: jedan ciklus je ponovljeni IF, druga dva su "prazni" ciklusi. Instrukcija iza grananja je dobavljena ali se ignoriše, a dobavljanje se ponavlja kada se ustanovi odredište grananja.

Poboljšanje se postiže na sljedeći način - potrebno je:

1. saznati ishod grananja ranije u PS-i, i
2. izračunati PC ranije (kod grananja sa ispunjenim uslovom).

Obje stvari se moraju obaviti što ranije u PS-i (slika 5.9.). Kako se u BEQZ i BNEZ testiraju/porede registri sa nulom, taj se posao može prebaciti u ID fazu. U oba slučaja jednog uslovnog grananja PC treba na vrijeme izračunati. To zahtjeva dodatni ALU (sabirač) u ID fazi jer se ne može čekati na glavnu ALU u EX fazi. Ovako se zastoj svodi na jedan ciklus kod grananja (tabela 5.8.).



Slika 5.9. Zastoj zbog hazarda grananja se može smanjiti premiještanjem testa nule i računanja odredišne adrese grananja u ID fazi protočne strukture.

Segment	Instrukcija grananja
IF	IF/ID.IR←Mem[PC];

	IF/ID.NPC, PC←(if (Regs[IF/ID.IR$_{6..10}$] *op* 0) {IF/ID.NPC + (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$}else {PC+4});
ID	ID/EX.A←Regs[IF/ID.IR$_{6..10}$]; ID/EX.B←Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR←IF/ID.IR; ID/EX.Imm←(IF.ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;
EX	
MEM	
WB	

Tabela 5.8. Prerađena protočna struktura koristi odvojeni sabirač za računanje odredišne adrese grananja za vrijeme ID. Nove i promjenjene operacije su podebljane. Izbor između inkrementiranog PC-a ili izračunatog PC kod grananja se i dalje dešava u IF, ali sada koristi vrijednost izID/EX, a ne EX/MEM registra.

Ostala rješenja kod grananja:

1. Najjednostavnije je riješiti grananje zastojima (freeze/flush) kao u tabeli 27.1 i sa SW i HW strane.
2. Bolji (brži) i nešto složeniji način je pretpostaviti da neće biti grananja - nastaviti kao da nema grananja. Ipak se ne smije mijenjati status mašine prije otkrivanja rezultata grananja. Ako se prihvati grananje, potrebno je "isprati" PS - pretvoriti donešene instrukcije u no-op-e (brisanjem IF/ID pregradnih registara) i restartati IF sa odredišne adrese (tabela 5.9.).
3. Pretpostaviti da će biti grananja - granati se čim se odredi odredišna adresa. Kod ogledne PS, ciljna adresa je poznata kada i rezultat ispitivanja uslova grananja pa ovo nema smisla. Kod mašina kod kojih se adresa može ranije izračunati, ovo bi moglo biti ubrzanje.
4. "Odgođeno grananje" (eng. delayed branch) (tabela 5.10.) iza instrukcije grananja ubacuje "instrukciju zadrške" (odgode grananja, eng. branch delay instruction), koja se svakako izvršava, dok se ne odluči o grananju (šta ako je i ubačena instrukcija grananje?! - ovo je zabranjeno i o tome kompajleri vode računa, danas sve mašine sa odgođenim grananjem imaju jednu instrukciju zadrške).

Grananje koje se ne desi	IF	ID	EX	MEM	WB				
Instrukcija i+1		IF	ID	EX	MEM	WB			
Instrukcija i+2			IF	ID	EX	MEM	WB		
Instrukcija i+3				IF	ID	EX	MEM	WB	
Instrukcija i+4					IF	ID	EX	MEM	WB

Grananje koje se desi	IF	ID	EX	MEM	WB				
Instrukcija i+1		IF	**idle**	**idle**	**idle**	**idle**			
Odredište grananja			IF	ID	EX	MEM	WB		
Odredište grananja +1				IF	ID	EX	MEM	WB	
Odredište grananja +2					IF	ID	EX	MEM	WB

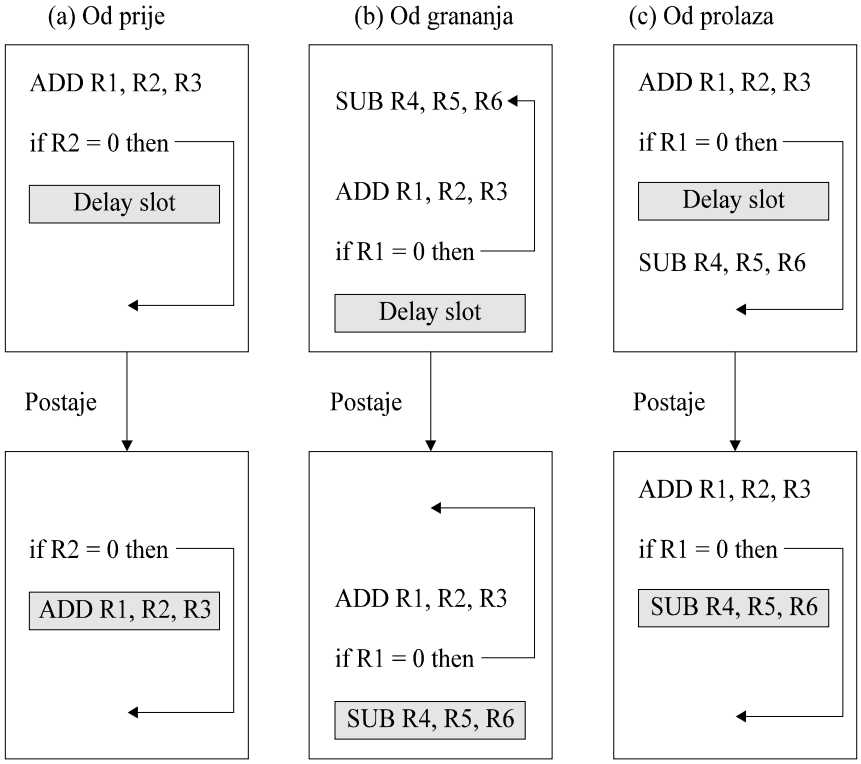
Tabela 5.9. Slučaj kada se predviđa da neće biti grananja i sekvence protočne strukture kada se grananje desi (gore) i ne desi (dole). Kada se grananje ne desi, što je određeno u ID, pribavljena je instrukcija iz prolazne grane i tako se izvršenje nastavlja. Ako se desi grananje tokom ID, ponovo se pokreće pribavljanje instrukcije sa odredišta grananja. To izaziva zastoj svih instrukcija iza grananja, za jedan ciklus.

Grananje koje se ne desi	IF	ID	EX	MEM	WB			
Ubačena instrukcija (i+1)		IF	ID	EX	MEM	WB		
Instrukcija i+2			IF	ID	EX	MEM	WB	
Instrukcija i+3				IF	ID	EX	MEM	WB

Instrukcija i+4					IF	ID	EX	MEM	WB

Grananje koje se desi	IF	ID	EX	MEM	WB				
Ubačena instrukcija (i+1)		IF	ID	EX	MEM	WB			
Odredište grananja			IF	ID	EX	MEM	WB		
Odredište grananja +1				IF	ID	EX	MEM	WB	
Odredište grananja +2					IF	ID	EX	MEM	WB

Tabela 5.10. Ponašanje odgođenog grananja je isto bez obzira da li se grananje desilo ili ne. Izvrši se ubačena instrukcija (postoji samo jedan "delay slot" kod ogledne PS). Ako se grananje ne desi, nastavlja se izvršenje iza ubačene instrukcije. Ako se desi grananje, izvršenje se nastavlja na odredištu grananja.



Slika 5.10. Raspoređivanje - biranje instrukcije za ubacivanje iza grananja. Gornji dio prikazuje kod prije, a donji nakon ubacivanja instrukcije. U (a) ubačena je nezavisna instrukcija iz dijela prije grananja. To je najbolje rješenje. Kada to nije moguće, koriste se (b) i (c). Tada korištenje R1 kao uslova grananja sprječava prebacivanje ADD (čije je odredište R1) iza grananja. Kod (b) se ubacuje instrukcija sa odredišta grananja. Ovo je dobro kada postoji velika vjerovatnoća da će se grananje desiti. Kod (c) se ubacuje instrukcija iz prolazne grane. Da bi obje optimizacije, (b) i (c), bile korektne mora se obezbijediti da izvršenje SUB, kada grananje ode nepredviđenim smjerom, ne pretstavlja problem - gubi se vrijeme ali ne utiče na korektnost izvršenja programa.

Zadatak kompajlera je da odabere instrukciju za ubacivanje iza grananja. Ovakva optimizacija je moguća na 3 načina (slika 5.10.). Najbolje je (a) ako je moguće naći pogodnu "neutralnu" instrukciju za zadršku. Ako nije (b, c) ubacuje se instrukcija koja ima veće šanse za izvršenje uz predviđanje grananja. Izvršenje SUB kad se grana u neočekivanom pravcu je gubitak vremena ali se program mora nastaviti bez problema.

Tabela 5.11. daje ograničenja kod ovakve optimizacije. Ograničenja kod metode odgođenog grananja zavise od:

1. ograničenja na instrukciju zadrške - koja ispunjava "delay slot" i

2. mogućnosti kompajlera da odredi vjerovatnoću prihvatanja/neprihvatanja grananja (statistički).

Raspoređivanje	Zahtjevi	Povećava performanse kada?
(a) Od prije grananja	Grananje ne smije zavisiti od preraspodijeljenih instrukcija.	Uvijek.
(b) Sa odredišta	Mora se moći izvršiti preraspodijeljene instrukcije ako se grananje ne desi. Može zahtijevati dupliciranje instrukcija.	Kada se grananje desi. Pože povećati program ako se instrukcije dupliciraju.
(c) Iz prolazne grane	Mora se moći izvršiti preraspodijeljene instrukcije ako se grananje desi.	Kada se grananje ne desi.

Tabela 5.11. Različiti načini raspoređivanja instrukcija kod odgođenog grananja i njihovi zahtjevi. Porijeklo ubačene instrukcije određuje strategiju raspoređivanja. Kada se ne može ubaciti ništa drugo - ubacuje se NOP instrukcija. Kada se (b) odredištu grananja može pristupiti i sa drugog mjesta u programu, odredišna instrukcija se kopira, a ne samo prebacuje.

Grananje koje se ne desi	IF	ID	EX	MEM	WB				
Ubačena instrukcija (i+1)		IF	ID	**idle**	**idle**	**idle**			
Instrukcija i+2			IF	ID	EX	MEM	WB		
Instrukcija i+3				IF	ID	EX	MEM	WB	
Instrukcija i+4					IF	ID	EX	MEM	WB

Grananje koje se desi	IF	ID	EX	MEM	WB				
Ubačena instrukcija (i+1)		IF	ID	EX	MEM	WB			
Odredište grananja			IF	ID	EX	MEM	WB		
Odredište grananja +1				IF	ID	EX	MEM	WB	
Odredište grananja +2					IF	ID	EX	MEM	WB

Tabela 5.12. Uz pretpostavku da će se grananje desiti, otkazivanje zavisi od rezultata grananja. Ubačena instrukcija se izvršava samo ako se grananje desi, inače se pretvara u NOP.

Da bi se omogućilo kompajlerima poboljšano popunjavanje instrukcija zadrške (eng. branch delay slotova - BDS) većina mašina sa BDS-ovima podržava otkazivanje/anuliranje grananja. BDS se popuni instrukcijom iz očekivanog pravca grananja. U slučaju pogotka - nema problema. U slučaju promašaja - grananju na neočekivanu stranu, instrukcija zadrške (iz BDS-a) se otkazuje/anulira - pretvara u no-op. Tabela 5.12. prikazuje slučaj pretpostavke da će se grananje desiti.

Mehanizam otkazivanja grananja omogućava olakšane preduslove odabiranja instrukcija zadrške - omogućava kompajlerima lakše korištenje (b) i (c) načina optimizacije (slika 5.10. i tabela 5.11.).

Komplikacije mogu nastati kod odgođenog grananja ako se desi prekid u trenutku izvršavanja instrukcije zadrške. U slučaju grananja, adresa instrukcije zadrške i adresa grananja nisu susjedne - za kasnije restoriranje konteksta PS-e su potrebna dva PC-a.

5.6.1. Statičko predviđanje grananja

Učinak odgođenog grananja djelimično zavisi od toga može li se pretpostaviti/predvidjeti na koju stranu će grananje ići. U skladu s tim kompajlerom se može smanjiti uticaj i upravljačkih hazarda.

Primjer:

```
        LW    R1, 0(R2)
        SUB   R1, R1, R2
        BEQZ  R1, L
        OR    R4, R5, R6
        . . .
L:      ADD   R7, R8, R9
```

Kako SUB i BEQZ zavise od LW, potreban je zastoj iza LW. Ako se zna da se grananje, u glavnom, dešava a da R7 nije potreban ako se ne desi grananje, program se može ubrzati prebacivanjem ADD iza LW. Ako se zna da se grananje rijetko dešava, a R4 nije potrebno nakon grananja, ubrzanje se postiže prebacivanjem OR iza LW.

Ovako se može rekonfigurisati svako odgođeno grananje, u skladu sa poznavanjem njegovog ponašanja.

Postoje dva osnovna načina statičkog predviđanja:
1. ispitivanjem ponašanja programa, i
2. korištenjem informacija prikupljenih prilikom ranijih izvršavanja programa.

Primjer: Neka program pokazuje prosječno 0.06 zastoja zbog grananja po instrukciji i 0.05 zastoja nakon LOAD-a po instrukciji, to (bez zastoja sa memorijom) dovodi do povećanja sa 1 CPI na 1+0.05+0.06=1.11 CPI, što i bez dodatnih kašnjenja clock-a u PS-i (idealno) daje smanjenje ubrzanja izvršenja programa pomoću PS-e sa 5 na 4.5 puta (5/1,11=4.5).

5.7. Prekidi/izuzeci u protočnoj strukturi

Izuzeci/prekidi komplikuju paralelizam u izvršavanju instrukcija u PS-i. Instrukcije se izvršavaju u dijelovima. Instrukcija u PS-i može izazvati izuzetak/prekid i prekid izvršavanja ostalih instrukcija u PS-i.

Situacije nastanka prekida/izuzetaka:
1. zahtjev U/I uređaja,
2. zahtjev korisničkog programa za servisom OS-a,
3. izvršavanje instrukcija korak-po-korak (eng. tracing),
4. programerski prekidi (eng. breakpoints),
5. aritmetičke int. i FP anomalije (prekoračenja, podbačaji …),
6. promašaj stranice virtuelne memorije,
7. neporavnat pristup memoriji,
8. pokušaj pristupa zaštićenom dijelu memorije (eng. memory protection violation),
9. nedefinisana instrukcija,
10. HW - greške,
11. greška napajanja

Izuzeci i prekidi se mogu podijeliti u sljedeće kategorije:

1. **Sinhroni/asinhroni**

Sinhroni su događaji koji se dešavaju u programu na istom mjestu sa istim podacima. Asinhroni (osim HW-problema) su izazvani vanjskim uređajima ili događajima i obrađuju se nakon završetka tekuće instrukcije - što ih čini lakšim za obradu.

2. **Traženi/prinudni**
Nekada se traženi i ne zovu izuzecima jer su predvidivi, ali koriste iste metode čuvanja i restoriranja (vraćanja) konteksta. Prinudne izazivaju HW događaji van kontrole korisničkog programa. Teško se obrađuju jer su nepredvidivi.

3. **Prekidi koji se mogu/ne mogu korisnički maskirati**
Manje važni događaji se mogu maskirati - zabraniti da izazivaju prekide. Ako su maskirani HW na njih ne reaguje.

4. **Unutar/između instrukcija**
Ako događaj onemogućava završetak počete instrukcije, prekid se zove unutarinstrukcijski. Ovi prekidi su uvjek sinhroni jer ih instrukcije "okidaju" - teško se obrađuju jer se izvršenje instrukcije mora prekinuti pa kasnije restartati. Ostali se "prepoznaju" između instrukcija i jednostavniji su za obradu.

5. **Nastavljivi/završni** (eng. resume/terminate)
Ako se program nastavlja nakon obrade izuzetka/prekida - on se zove nastavljivi. Ako se prekida izvršenje programa - prekid se zove završni (lakše se realizuju jer nema restarta programa).

Tabela 5.13. da je prikaz različitih imena za ove slučajeve kod različitih arhitektura. Tabela 5.14. klasificira primjere iz tabele 5.13. na ovih pet kategorija.

Događaj - izuzetak	IBM 360	VAX	Motorola 680x0	Intel 80x86
Zahtjev U/I uređaja	U/I prekid	Prekid od uređaja	Izuzetak (Nivo 0...7 autovektor)	Vektorisani prekid
Poziv OS-servisa od strane korisničkog programa	Prekid zbog supervizor poziva	Izuzetak (promjena mode-a supervizor trap)	Izuzetak (nepostojeća instrukcija) - na Macintosh-u	Prekid (INT instrukcija)
Praćenje izvršenja instrukcija	Ne postoji	Izuzetak (trace greška)	Izuzetak (trace)	Prekid (korak-po-korak trap)
Prekidna tačka	Ne postoji	Izuzetak (greška prekidne tačke)	Izuzetak (ilegalna instrukcija ili prekidna tačka)	Prekid (trap prekidne tačke)
Prekoračenje ili podbačaj kod cjelobrojne aritmetike; FP trap	Prekid izvođenja programa (izuzetak prekoračenja ili podbačaja)	Izuzetak (cjelobrojno prekoračenje trap ili greška FP podbačaja)	Izuzetak (greške FP-koprocesora)	Prekid (trap prekoračenja ili izuzetak FPU-a)
Greška stranice u memoriji (nije u glavnoj memoriji)	Ne postoji (samo kod 370)	Izuzetak (greška kod translacije)	Izuzetak (greška MMU)	Prekid (greška stranice)
Neporavnati pristup memoriji	Prekid izvođenja programa (specification exception)	Ne postoji	Izuzetak (greška adrese)	Ne postoji
Pristup zabranjenoj zoni u memoriji	Prekid izvođenja programa (protection exception)	Izuzetak (greška -narušavanje prava pristupa)	Izuzetak (greška na sabirnici)	Prekid (protection exception)
Korištenje nedefinisane instrukcije	Prekid izvođenja programa (operation exception)	Izuzetak (opcode privileged/reserved fault)	Izuzetak (ilegalna instrukcija ili prekidna tačka)	Prekid (nekorektan opkod)
Hardverske greške	Machine-check prekid	Izuzetak (machine-check abort)	Izuzetak (greška na sabirnici)	Ne postoji
Greška napajanja	Machine-check prekid	Hitni prekid	Ne postoji	NMI

Tabela 5.13. Imena uobičajenih izuzetaka u četiri različite arhitekture. Svaki događaj kod IBM 360 i 80x86 se zove prekid (*interrupt*), dok se kod 680x0 zove izuzetak (*exception*). U isto vrijeme, VAX dijeli događaje na prekide i izuzetke.

Zbog neophodnosti podrške virtuelnoj memoriji, većina savremenih PS-a je restartabilna. Kod ogledne arhitekture, greška pri pristupu adresi u virtuelnoj memoriji se otkriva u MEM fazi kada je više drugih instrukcija već u fazi izvršenja - PS treba bezbjedno zaustaviti (ugasiti) i stanje/kontekst sačuvati, kako bi se, nakon obrade izuzetka, mogla restorirati u originalno stanje. Ako instrukcija od koje treba nastaviti nije grananje, dovoljno je sačuvati PC. U protivnom treba ponovo ispitati uslov grananja, što komplikuje slučaj.

U trenutku prekida je potrebno:

1. ubaciti TRAP instrukciju u IF PS-e,
2. zabraniti sve upise pri izvršavanju zatečenih instrukcija u PS-i,
3. pozvana rutina za obradu izuzetaka treba sačuvati PC instrukcije koja je izazvala prekid.

Međutim, kod odgođenog grananja nije dovoljan jedan PC za restoriranje stanja (jer

instrukcije u PS-i ne moraju biti uzastopne), već onoliko PC-a kolika je dubina/dužina odgođenog grananja +1. Ovo se obavlja u pozvanoj rutini (3 korak).

Ako se PS može zaustaviti tako da su instrukcije prije one koja izaziva izuzetak - obavljene a one poslije se mogu restartati od početka.

Tip izuzetka	Sinhrono ili asinhrono	Traženi ili prinudni	Mogu se maskirati ili ne	Unutar ili između instrukcija	Prolazni ili neprolazni
Zahtjev U/I uređaja	Asinhrono	Prinudni	Ne mogu	Između	Prolazni
Poziv OS-a	Sinhrono	Traženi	Ne mogu	Između	Prolazni
Praćenje izvršenja instrukcija	Sinhrono	Traženi	Mogu	Između	Prolazni
Prekidna tačka	Sinhrono	Traženi	Mogu	Između	Prolazni
Prekoračenje kod cjelobrojne aritmetike	Sinhrono	Prinudni	Mogu	Unutar	Prolazni
Prekoračenje ili podbačaj kod FP aritmetike	Sinhrono	Prinudni	Mogu	Unutar	Prolazni
Greška stranice u memoriji	Sinhrono	Prinudni	Ne mogu	Unutar	Prolazni
Neporavnat pristup memoriji	Sinhrono	Prinudni	Mogu	Unutar	Prolazni
Narušavanje memorijske zaštite	Sinhrono	Prinudni	Ne mogu	Unutar	Prolazni
Nedefinisana instrukcija	Sinhrono	Prinudni	Ne mogu	Unutar	Neprolazni
Greška hardvera	Asinhrono	Prinudni	Ne mogu	Unutar	Neprolazni
Greška napajanja	Asinhrono	Prinudni	Ne mogu	Unutar	Neprolazni

Tabela 5.14. Klasifikacija primjere iz tabele na slici 4.23 na pet kategorija.

Bilo bi idealno da instrukcija koja izaziva grešku/izuzetak ne mijenja stanje mašine. Međutim, npr. FP-instrukcije često pišu svoje rezultate prije obrade izuzetka. Tada hardver mora biti u stanju pribaviti izvorne (source) operande (FP-operacije mogu trajati više ciklusa - neke druge instrukcije mogu pisati po izvornim operandima, pogotovo u slučaju izvršavanja instrukcija mimo reda). Zato neke mašine (Alpha, Power-2, MIPS) uvode dva načina rada - sa i bez precizne obrade izuzetaka, pri čemu je razlika u brzini i više od 10 puta (zbog manjeg dozvoljenog preklapanja među FP-operacijama).

5.7.1. Izuzeci kod ogledne arhitekture

Tabela 5.15. daje segmente ogledne PS-e i moguće “probleme” koji mogu izazvati izuzetke u svakom od njih. Par instrukcija LW i ADD može izazvati izuzetak kod pristupa stranici podataka u virtuelnoj memoriji i izuzetak kod aritmetičke operacije istovremeno jer je LW u MEM segmentu u trenutku kada je ADD u EX segmentu. Ovo se može prevazići obradom prvog izuzetka, a zatim restartanjem izvršenja. Drugi izuzetak će se ponovo desiti i može se obraditi nezavisno od prvog.

Segment PS-e	Problemi i izuzeci koji se javljaju
IF	Greška stranice u memoriji pri dobavljanju instrukcije; neporavnat pristup memoriji; narušavanje zaštite memorije

ID	nedefinisan ili ilegalni opkod
EX	Aritmetički izuzetak
MEM	Greška stranice u memoriji pri dobavljanju podatka; neporavnat pristup memoriji; narušavanje zaštite memorije
WB	Ne postoje

Tabela 5.15. Izuzeci koji se mogu javiti u oglednoj PS-i. Izuzeci kod pristupa memoriji (instrukcija ili podataka) predstavljaju šest od osam slučajeva.

U praksi nije sve tako jednostavno. Izuzeci se mogu pojaviti i mimo reda, npr. LW može izazvati grešku u pristupu memoriji podataka u MEM fazi a ADD može izazvati grešku u pristupu memoriji instrukcija (page fault) kada je u IF fazi.

Pošto treba realizovati preciznu obradu izuzetaka, od PS-e se očekuje da prvo obradi LW izuzetak. Neka je LW i-ta instrukcija a ADD i+1-va. PS ne može jednostavno izvršavati obrade izuzetaka kada se oni dese jer bi se mogla desiti obrada mimo reda izvršenja instrukcija. Umjesto toga HW označi sve izuzetke izazvane datom instrukcijom u status-vektoru koji joj pridruži i koji je prati kroz PS-u. Tada se onemogućava svaki upis podataka koji bi ta instrukcija trebala obaviti (bilo u registre ili u memoriju). Upis u MEM može izazvati izuzetak, pa hardver mora spriječiti završetak ove instrukcije u tom slučaju.

Kada instrukcija dođe u WB fazu (napušta MEM) provjerava se status vektor izuzetaka. Ako je označen ijedan izuzetak, oni se obrađuju redoslijedom kojim bi nastajali u mašini bez PS-e. Tako se obezbjeđuje da se svi izuzeci uz i-tu instrukciju vide prije onih od i+1-ve. Sve što je i-ta instrukcija već uradila (ili je u njeno ime urađeno) ne važi, ali kako su upisi (rezultata) onemogućeni (u registre i memoriju) nije moglo biti promijenjeno stanje mašine. Ovo je znatno teže izvesti u operacijama sa pomičnim zarezom.

5.8. Komplikacije skupa instrukcija

Svaka instrukcija ogledne arhitekture ima samo jedan rezultat, i upisuje se samo na kraju izvršenja. Kod cjelobrojne ogledne PS-e instrukcija se može smatrati izvršenom (eng. commited) na kraju MEM - početku WB i nijedna ne mijenja stanje mašine prije toga. Time je olakšana precizna obrada izuzetaka.

Neke mašine imaju instrukcije koje mijenjaju stanje na sredini izvršenja, prije nego što je sigurno da će se ta (i prethodna) instrukcija završiti (VAX!!!). Bez dodatnog HW-a nije moguće realizovati preciznu obradu izuzetaka - teško je restartati izvršenje programa.

Na primjer, LOOP/autoincrement instrukcije mijenjaju sadržaj nekog registra više puta u toku svog izvršenja. Da bi se izvela precizna obrada prekida, mora postojati mehanizam za neutralisanje (back out) promjena stanja prije nego što se instrukcija kompletira (međurezultati se mogu sačuvati na stack, ali šta ako se npr. kopiraju stringovi u memoriji?).

Problemi PS-e su očigledni kod višeciklusnih operacija. Npr. kod VAX-a:

```
MOV     R1, R2
ADDL3   42(R1), 56(R1)+, @(R1)
SUBL2   R2, R3
MOVC3   @(R1)[R2], 74(R2), R3
…
```

Izvršenje ovakvih instrukcija traje od 1 do 100 ciklusa i zahtijevaju različit broj pristupa memoriji (od 0 do više stotina). Ovdje je izražen problem npr. hazarda podataka (između i

unutar instrukcija). Svođenje izvršenja ovakvih instrukcija na isti broj ciklusa je gotovo nemoguće (gubici, predugačka PS-a, …). Kod VAX-a 8800 mikroinstrukcije se izvode u PS-i (sličnoj oglednoj!!!).

5.9. Ogledna arhitektura i višeciklusne operacije

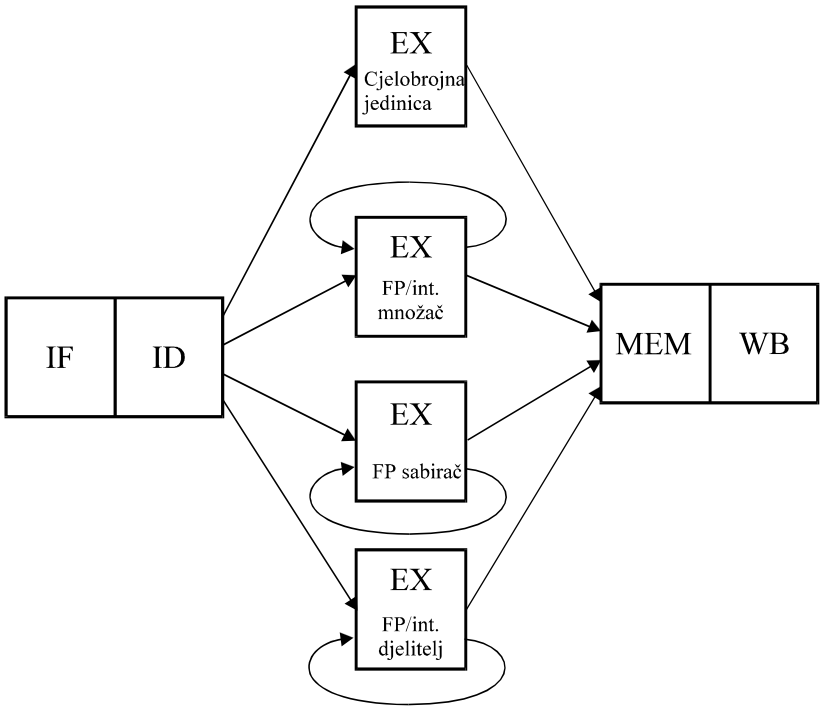
Nepraktično je zahtijevati izvršenje FP-operacija u 1 do 2 ciklusa sata. To bi produžilo cikluse ili izkomplikovalo neophodan hardver (ili oboje). Jedno od rješenja je da se EX ciklus/faza može ponavljati željeni broj puta, i da postoji više odvojenih FP-funkcionalnih jedinica (FJ). Zastoji bi nastajali u slučaju strukturnih ili hazarda podataka. Na primjer, slika 4.27 prikazuje 4 odvojene FJ-e u oglednoj arhitekturi. Neka EX jedinice nisu realizovane kao PS-e. Tada nijedna instrukcija ne može početi izvršenje dok se prethodna ne završi (čitava PS-a prije EX će biti u zadršci).

U realnosti, međurezultati najčešće ne cirkuliraju oko EX jedinica kao na slici 5.11., već EX ima PS-u sa kašnjenjem većim od jedan ciklus. U opštem slučaju može biti više takvih PS-a i više paralelnih operacija. Tada treba definisati kašnjenje u funkcionalnoj jedinici (eng. latency) i interval ponavljanja - pokretanja. **Kašnjenje**, kao broj ciklusa između instrukcije koja proizvodi rezultat i instrukcije koja ga koristi. **Period ponavljanja**, kao broj ciklusa između dva uzastopna završetka operacija datog tipa. Primjer tih vrijednosti je dat u tabeli 5.16..

Funkcionalna jedinica	Kašnjenje	Interval pokretanja
Cjelobrojna ALU	0	1
Memorija podataka (cjelobrojnih i FP)	1	1
FP sabiranje	3	1
FP množenje (i cjelobrojno)	6	1
FP dijeljenje (i cjelobrojno dijeljenje i FP korjenovanje)	24	24

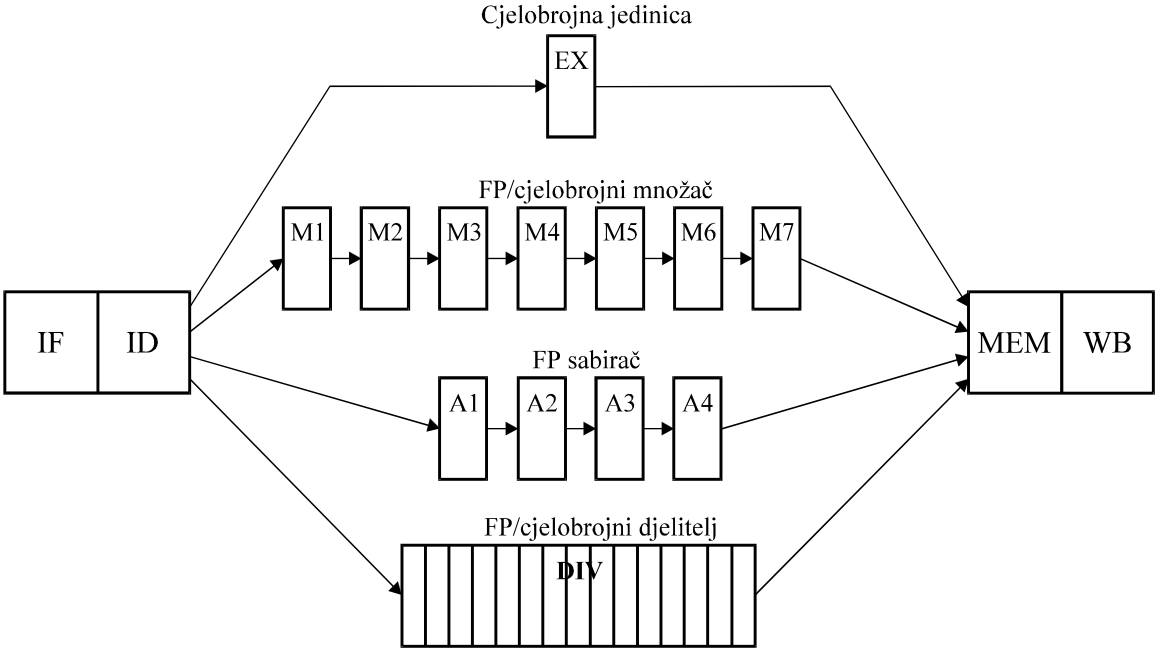
Tabela 5.16. Kašnjenja i intervali pokretanja funkcionalnih jedinica.

Po njima, cjelobrojna ALU ima kašnjenje 0 jer se njen rezultat može koristiti u sljedećem ciklusu, čitanja podataka (loads) imaju kašnjenje 1 jer se rezultat može koristiti nakon 1 ciklusa. Pošto većina instrukcija koriste svoje operande na početku EX-a, kašnjenje je obično jednako broju segmenata PS-e poslije EX-a, kada instrukcija daje rezultat - 0 za ALU, 1 za LOAD. Izuzetak su upisi u memoriju, koji koriste vrijednosti smještene jedan ciklus kasnije, pa će kašnjenje do smještaja vrijednosti (ali ne za base adres. reg.) biti jedan ciklus manje. Kašnjenje PS-e je jednako jedan ciklus manje od dubine EX PS-e, tj. broj segmenata od EX-a do segmenta koji proizvodi rezultat. Prema tome, broj segmenata u FP-sabiranja je 4, dok je kod FP-množenja 7. Da bi se postigle više frekvencije signala sata, broj nivoa logike u svakom segmentu se mora smanjiti (za složenije operacije će trebati više segmenata - brži signal sata - veća kašnjenja operacije). Na osnovu slike 5.11. i tabele 5.16., kada se broj intervala ponavljanja zamijeni dodatnim segmentima PS-e, dobije se slika 5.12. Samo jedna operacija može biti aktivna u jednom trenutku.



Slika 5.11. Ogledna PS sa tri dodatne neprotočne FP funkcionalne jedinice. Kako se svakog ciklusa pokreće samo jedna instrukcija, sve instrukcije idu kroz standardnu PS za cjelobrojne operacije. FP operacije se jednostavno vrte u petlji kada stignu u EX segment. Nakon završetka EX faze, one nastavljaju kroz MEM i WB da dovrše izvršenje.

Tabela 5.17. daje vremenski redoslijed skupa nezavisnih FP-operacija i FP-čitanja i FP-pisanja. Dužina kašnjenja FP-operacija povećava učestalost RAW-hazarda i odgovarajućih zastoja. Struktura PS-e sa slike 5.12. zahtijeva dodatne pregradne registre i modifikovane veze. ID sa sljedećim segmentom može vezati jedan ili više paralelnih registara. Kako u jednom segmentu može biti samo jedna operacija u jednom trenutku - instrukcija ide zajedno sa podacima na ulaz u svaki segment.



Slika 5.12. Protočna struktura koja podržava više istovremenih FP operacija. FP množač i sabirač su potpuno protočni, dubine sedam i četiri segmenta respektivno. FP djelitelj nije protočan, ali zahtijeva 25 ciklusa sata za obavljanje operacije. Kašnjenje u instrukciji između pokretanja FP operacije i korištenja rezultata te operacije, bez izazivanja RAW zastoja je određeno brojem ciklusa provedenih u EX segmentima. Primjer: četvrta instrukcija poslije FP sabiranja može koristiti taj rezultat. Za cjelobrojnu ALU operaciju, dubina EX protočne strukture je uvijek jedan, i sljedeća instrukcija može koristiti rezultat. FP i cjelobrojna čitanja iz memorije se završavaju u MEM fazi, što znači da memorijski sistem mora obezbijediti 32 ili 64 bita u jednom ciklusu sata.

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	*MEM*	WB			

Tabela 5.17. Slijed skupa nezavisnih FP operacija u PS-i. Stanja označena italikom pokazuju gdje su podaci potrebni, dok podebljane oznake pokazuju gdje su ti podaci raspoloživi. FP upisi u memoriju, kao i cjelobrojni, koriste svoje izvorne operande u dva različita trenutka.

Hazardi i proslijeđivanja kod dubokih PS-a:

1. Kako sklop za dijeljenje nije u potpunosti PS-a, mogu se javiti strukturni hazardi, pa ovo treba otkriti i ubaciti odgovarajuće zastoje.
2. Kako instrukcije imaju različita vremena izvršenja, broj zahjeva za upis u registre u jednom ciklusu može biti veći od jedan.
3. Mogući su WAW-hazardi jer instrukcije više ne ulaze u WB redom (WAR nije moguć jer se čitanje registara uvijek obavlja u ID-fazi).
4. Instrukcije se mogu izvršiti drugim redoslijedom od onog kojim su pokrenute, pa nastaju problemi sa izuzecima.
5. Zbog povećanog kašnjenja kod izvršavanja operacija, zastoji zbog RAW-hazarda će biti češći.

Moguće posljedice RAW-hazarda su vidljive iz tabele 5.18. sa tipičnom sekvencom FP-instrukcija i odgovarajućim zastojima. Kod (2) i (3) iz prethodne liste se javlja problem u slučaju da FP-skup registara ima jedan port za upis - tada sekvenca FP-operacija i FP-load-a mogu izazvati konflikt nad tim portom (strukturni hazard).

	Broj ciklusa sata														
Instrukcija	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD F4, 0(R2)	IF	ID	EX	MEM	WB										
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB		
ADDD F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4
SD F2, 0(R2)				stall	IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall

Tabela 5.18. Tipična sekvenca FP koda pokazuje zastoje (stall) nastale zbog RAW hazarda. Duže PS-e imaju višu učestalost pojave zastoja u odnosu na kratke cjelobrojne PS-e. Svaka instrukcija u ovoj sekvenci zavisi od prethodne i nastavlja sa izvršenjem čim su joj operandi raspoloživi, što podrazumijeva da PS ima mehanizme proslјeđivanja. Instrukcija SD mora biti odgođena još jedan ciklus tako da njegova MEM faza nije u konfliktu sa ADDD. Dodatni hardver može riješiti ovaj problem.

U tabeli 5.19. je data sekvenca u PS-i. U ciklusu sata br. 11, sve tri instrukcije će doći do WB i htjeti pisati u skup registara. Sa samo jednim portom za upis mora se serijalizirati - sekvencijalno završiti instrukcije. Povećanje broja portova za pisanje bi moglo riješiti problem, ali bi se dodatni portovi rijetko koristili. Umjesto toga potrebno je detektovati ovaj hazard i riješiti ga uobičajenom metodom.

Tu postoje dva rješenja.

Prvo je - pratiti korištenje porta za upis u ID-segmentu i zadržati instrukciju koja bi izazvala hazard (kao kod svakog strukturnog hazarda). To se može postići pomoću šift-registra u kome se označi kada će već pokrenuta (issued) instrukcija koristiti skup registara. Ako instrukcija u ID-fazi treba da koristi skup registara istovremeno kad i pokrenuta instrukcija - ona u ID-segmentu se zadržava jedan ciklus. Svakog ciklusa sata "registar rezervacije" se šifta za 1 bit. Prednost ovakvog pristupa je u tome što rješava problem "lokalno" u ID segmentu i "jeftino" je rješenje.

Alternativno rješenje je - zadržati konfliktnu instrukciju kada pokuša ući u MEM-segment - tada je moguće izabrati instrukciju koju treba zadržati. Pri tome je logično dati prednost onoj koja ima najveće kašnjenje, jer bi ona i dalje izazivala RAW-hazarde. Ovako se ne traži otkrivanje hazarda sve do MEM-segmenta, gdje je to lako uočiti. Nedostatak ovakvog pristupa je što usložnjava upravljanje PS-om - zastoji mogu doći sa dva mjesta.

	Broj ciklusa sata										
Instrukcija	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F2, 0(R2)							IF	ID	EX	MEM	WB

Tabela 5.19. Tri instrukcije hoće da izvrše upis nazad u FP registre istovremeno, kao što je prikazano u ciklusu 11. Još gori slučaj bi bio da se, ranije započeto, dijeljenje u FP jedinici završava u istom ciklusu.

Mogućnost pojave WAW-hazarda se vidi u tabeli 5.19. Da je LD pokrenuta jedan ciklus ranije i da je imala odredište u FP-registru F2, izazvala bi WAW-hazard jer bi pisala u F2 jedan ciklus prije ADDD. Ovo se može desiti samo ako rezultat ADDD-a niko ne koristi prije LD, inače bi došlo do zastoja zbog RAW-hazarda i LD se nebi pokrenula prije dovršetka ADDD (Diskusija - uprkos tome hazard treba otkriti).

WAW-hazard se rješava na dva načina. Prvi je - zadržati LD dok ADDD ne uđe u MEM-segment, a drugi - "suzbiti" rezultat ADDD otkrivanjem hazarda i promjenom upravljanja, tako da ADDD ne upiše svoj rezultat. Tada se LD može pokrenuti odmah. Ove stvari može rješavati i kompajler - uz detaljno poznavanje PS-e i stanja instrukcija u svakom trenutku!!!

Pri otkrivanju mogućih hazarda treba razmotriti iste između FP-instrukcija kao i FP i cjelobrojnih instrukcija. Sa izuzetkom FP-load/store i FP-int reg. move, cjelobrojne instrukcije rade samo sa cjelobrojnim registrima, dok FP-instrukcije rade samo sa FP-skupom registara. Zato treba samo paziti na te instrukcije pri detekciji hazarda između FP i cjelobrojnih instrukcija. Pod pretpostavkom da PS otkriva hazarde u ID-segmentu, postoje tri provjere prije pokretanja instrukcije:

1. Provjera strukturnog hazarda - treba sačekati da tražena funkcionalna jedinica prestane biti zauzeta (u ovoj PS-i to je sklop za dijeljenje) i obezbijediti da je port za upis u registre slobodan kada bude potreban.
2. Provjera RAW-hazarda podataka - čekati sve dok izvorišni registri više ne budu na "spisku" odredišta "na čekanju" (pending destinations).
3. Provjera na WAW-hazard podataka - provjeriti da li ijedna instrukcija u A1, …, A4, D, M1,…,M7, ima isti odredišni registar kao tekuća, ako ima, uvesti zastoj u pokretanju te instrukcije (u ID-segmentu).

Iako je otkrivanje hazadra složenije kod multi-ciklusnih FP-operacija, principi su isti kao kod cjelobrojne ogledne PS-e. Isto se odnosi i na logiku prosljeđivanja - provjerom da li su odredišni registri (polja u EX/MEM, A4/MEM, M7/MEM, D/MEM ili MEM/WB) ujedno izvorni registri neke FP-instrukcije. Ako jesu, dodatnim MUX-evima se vrši prosljeđivanje.

Višeciklusne FP-operacije usložnjavaju i mehanizam prekida, na primjer:

```
DIVF  F0, F2, F4
ADDF F10, F10, F8
SUBF  F12, F12, F14
```

je jednostavna sekvenca bez međuzavisnosti. Međutim može se desiti da se ranije pokrenuta instrukcija završi kasnije (ADDF i SUBF poslije DIVF). Izvršavanje mimo redoslijeda (out-of-order completion) je uobičajeno kod PS-a sa dugotrajnim operacijama.

Šta ako SUBF izazove izuzetak FP-aritmetike u trenutku kada je ADDF završena a DIVF nije? Nastao bi neprecizni izuzetak jer se ne može vratiti staro stanje registara koje je ADDF uništio!!!

Tada postoje 4 rješenja:

1. Ignorisati problem i nastaviti sa nepreciznim obradama izuzetaka. To se radilo 60-tih i 70-tih, a i danas se radi kod nekih superkompjutera gdje određene vrste izuzetaka nisu dozvoljene ili se rješavaju hardverski bez zaustavljanja PS-e. Danas je teško ovako raditi zbog mehanizama (npr. virtuelne memorije, IEEE FP-operacija) koji zahtijevaju precizne izuzetke kroz hardver ili softver. Neke savremene mašine imaju dva načina rada - brzi/neprecizni i sporiji/precizni. Sporiji se svodi na ubacivanje eksplicitnih instrukcija koje testiraju FP-izuzetke. U tom načinu rada smanjeno je preklapanje i izvršavanje mimo reda u FP PS-i (efektivno samo jedna FP-instrukcija je aktivna u svakom trenutku. Tako rade DEC Alpha 21064 i 21164, IBM Power-2 i MIPS 8000.
2. Baferovanje rezultata dok se ranije pokrenute instrukcije ne završe. Ovo postaje skupo kada su vremena izvršavanja značajno različita, jer se tada povećava broj baferovanih rezultata (veliki broj komponenata, veliki MUX-evi …). Postoje dvije varijacije ovog rješenja. Prva je - korištenje zapisa o istoriji događaja (“history file” kod CYBER 180/990). Pamte se stanja registara, koja se u slučaju pojave izuzetka vraćaju/obnavljaju (rollback). Druga je - zapisivanja novih sadržaja registara, kojim se, nakon izvršenja prethodnih instrukcija, osvježavaju glavni registri. U slučaju pojave izuzetaka, glavni skup registar sadrži precizne vrijednosti prekinutog stanja (“future files” kod POWER PC 620, MIPS R10000).
3. Dozvoliti da izuzeci postanu “nešto malo” neprecizni ali zadržavati dovoljno informacija da se (trap handling rutinama) može restaurirati precizna originalna sekvenca za obradu izuzetka. To znači znati koje su operacije bile u PS-i u njihove PC-e. Zatim, nakon obrade izuzetka, softver završava instrukciju koja prethodi zadnjoj završenoj instrukciji i sekvenca se može ponovo pokrenuti.

   Primjer:

instr. 1	- dugotrajna instrukcija koja na kraju prekida izvršenje,
instr. 2 … n-1	- serija nedovršenih instrukcija,
instr. n	- završena instrukcija.

Poznavanjem PC-a svih instrukcija u PS-i i povratni PC izuzetka, softver može naći stanje instr. 1 i instr. n. Pošto je instr. n završena, potrebno je izvršenje nastaviti/restartati od instr. n+1. Poslije obrade prekida, softver mora simulirati izvršenje instr. 1 … instr. n-1. Tada je moguć povratak iz izuzetka i nastavak izvršenja od instr. n+1 (SPARC).

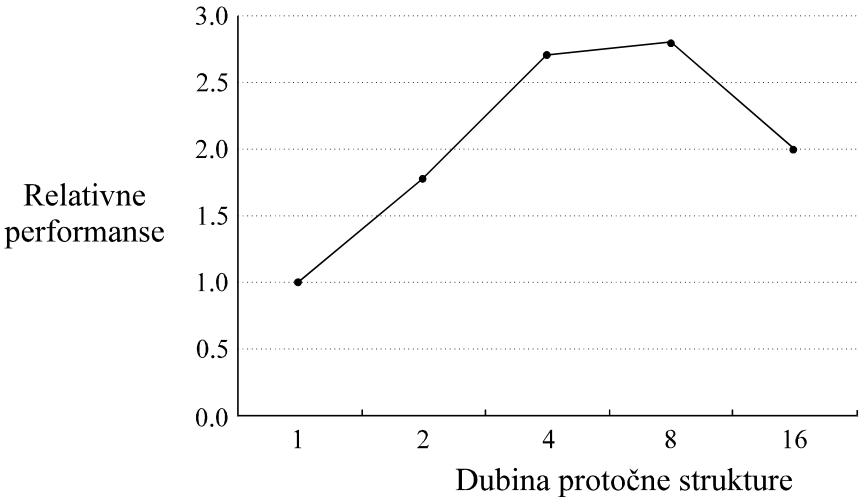
4. Hibridna tehnika - dozvoljava pokretanje novih instrukcija tek kada je sigurno da će se sve prethodne izvršiti bez izazivanja izuzetaka. Tako je osigurano da, kada se desi izuzetak, neće se izvršiti nijedna instrukcija pokrenuta poslije one koja je izazvala izuzetak (MIPS 2000/3000, R4000, Intel Pentium).

Česta se pogrešno smatra da neočekivana sekvanca instrukcija može izazvati neočekivane hazarde. Na prvi pogled WAW-hazard nema puno smisla, kao i dva upisa u registar bez čitanja između njih! Međutim, kod neočekivane sekvence, to je moguće, na primjer:

```
        BNEZ R1, foo
        DIVD   F0, F2, F4     ; ubačena instrukcija za odgođeno grananje
        …                     ; iz nastavka bez grananja
foo:    LD     F0, qrs
```

Ako se desi grananje, LD će stići u WB prije završetka DIVD, izazivajući WAW. Hardver ovo mora otkriti i zadržati pokretanje LD.

Ne može se reći da povećanje dubine PS-e uvijek povećava i performanse. Zbog međuzavisnosti i zastoja raste CPI. Kašnjenje clock-a (distribucija na čipu) i kašnjenja u latch-evima limitiraju porast frekvencije sata. Trend performansi u zavisnosti od broja segmenata PS-e je dan na slici



Slika 5.13. Dubina PS i njen uticaj na postignuto ubrzanje. X-osa pokazuje broj segmenata u EX dijelu FP protočne strukture. Jednosegmentna PS odgovara 32 nivoa logičkih kola, što može odgovarati jednoj FP operaciji. Istina je da su se na tržištu pojavili procesori sa PS-om sa više od 20 segmenata, ali je naknadna pojava višejezgrenih procesora potvrdila trend na slici.

6. Paralelizam na nivou instrukcija

Paralelizam u izvršenju instrukcija se odvija u PS-i ako instrukcije nisu međusobno zavisne. Statistički 15% instrukcija u cjelobrojnim operacijama su grananja (dinamička), pa se 6 do 7 instrukcija obavi u osnovnim blokovima između dva grananja. To umanjuje mogućnost preklapanja u izvršenju instrukcija. Da bi se postigle veće performanse, neophodno je iskoristiti paralelizam na nivou instrukcija (Instruction-Level Parallelism - ILP) u više osnovnih blokova istovremeno.

Primjer: paralelizam na nivou petlje (LLP)

```
for (i=1; i<=1000; i=i+1)
    x[i]=x[i]+y[i];
```

Prolazi kroz petlju se međusobno mogu preklapati - unutar jednog prolaza je teško postići preklapanje.

Pretvaranje LLP u ILP se postiže “odpetljavanjem” / ”odmotavanjem” petlje i to na dva načina - statički/kompajlerom i dinamički/hardverom.

Neka su kašnjenja FP jedinica data u tabeli 6.1., a cjelobrojna ogledna PS-a ima kašnjenje kod grananja u trajanju od jednog ciklusa. Neka su sve funkcionalne jedinice realizovane kao protočne ili replicirane dovoljan broj puta, tako da se instrukcije pokreću svakog ciklusa sata i nema strukturnih hazarda.

Instrukcija koja daje rezultat	Instrukcija koja koristi rezultat	Kašnjenje u ciklusima
FP ALU op	Druga FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Tabela 6.1. Kašnjenje FP operacija korištenih u ovom poglavlju. Prva kolona pokazuje tip izvorišnih instrukcija. Druga kolona pokazuje tip instrukcije korisnika rezultata. Zadnja kolona daje broj ubačenih ciklusa sata, potrebnih da bi se izbjegli zastoji. Ovi brojevi su slični prosječnim kašnjenjima kod ranije date FP jedinice. Glavna razlika je smanjeno kašnjenje FP množenja. Kašnjenje između FP-čitanja i pisanja je nula, jer se rezultat čitanja može proslijediti na upis bez zastoja. Kašnjenje cjelobrojnog čitanje ostaje 1 a cjelobrojne ALU operacije nula.

Kako kompajler može povećati ILP odmotavanjem sljedeće petlje?

```
for (i=1; i<=1000; i++)
    x[i]=x[i]+s;
```

Iteracije su nezavisne jedna od druge jer je tijelo petlje nezavisno. Neka R1 sadrži adresu elementa u nizu sa najvišom adresom (najniža je 0 - radi jednostavnosti primjera), F2 sadrži skalar s.

Direktni prevod u asembler, bez prilagođavanja PS-i je:

```
Loop: LD      F0, 0(R1)
      ADDD    F4, F0, F2
      SD      0(R1), F4
```

```
SUBI    R1, R1, #8
BNEZ    R1, Loop
```

Prema podacima iz tabele 6.1, ovo bi se izvodilo po ciklusima kao:

			Ciklus
Loop:	LD	F0,0(R1)	1
	Zastoj		2
	ADDD	F4, F0, F2	3
	Zastoj		4
	Zastoj		5
	SD	0(R1), F4	6
	SUBI	R1, R1, #8	7
	Zastoj		8
	BNEZ	R1, Loop	9
	Zastoj		10

Promjenom redoslijeda instrukcija na:

```
Loop: LD      F0, 0(R1)
      Zastoj
      ADDD    F4, F0, F2
      SUBI    R1, R1, #8
      BNEZ    R1, Loop      ; odgođeno grananje i modifikovana instrukcija
      SD      8(R1), F4     ; (bilo je 0(R1)) i zamjena SD i SUBI
```

smanjuje vrijeme izvršenja sa 10 na 6 ciklusa. Za ovakve zahvate kompajler mora biti "pametan" da zamijeni SUBI i SD i promijeni adresu upisa SD-instrukcije. Od ovih 6 ciklusa, 3 su korisna (LD, ADDD i SD) a 3 su petlja (SUBI i BNEZ) i zastoj. Za dalje ubrzanje neophodno je odmotati petlju - replicirati petlju više puta i podesiti broj prolaza i izlaska iz petlje. Odmotavanje olakšava raspoređivanje instrukcija. Uzastopno korištenje istog/istih registara može usporiti izvršenje. Promjenom koda - korištenjem više registara postiže se još veći ILP (uz povećanje potrebnog broja registara).

Primjer: Neka je R1 multipl od 32 a broj iteracija petlje multipl od 4. Izbjegavanjem više korištenja istog registra u jednom prolazu kroz "odmotanu" petlju, dobije se:

```
Loop: LD      F0, 0(R1)
      ADDD    F4, F0, F2
      SD      0(R1), F4     ; preskače se SUBI i BNEZ
      LD      F6, -8(R1)
      ADDD    F8, F6, F2
      SD      -8(R1), F8    ; preskače se SUBI i BNEZ
```

```
LD      F10, -16(R1)
ADDD    F12, F10, F2
SD      -16(R1), F12        ; preskače se SUBI i BNEZ
LD      F14, -24(R1)
ADDD    F16, F14, F2
SD      -24(R1), F16
SUBI    R1, R1, #32
BNEZ    R1, Loop
```

Uštađeno je 3 grananja i dekrementiranja R1.

Bez odgovarajućeg raspoređivanja, svaka operacija je praćena zavisnom, pa bi jedan prolaz trajao 27 ciklusa (LD-2, ADDD-3, BNEZ-2, ostale 1) ili 6,4 cilusa po svakoj kopiji originalne petlje - sporije nego ranije. Povećanje potrošnje memorijskog prostora je očigledno.

Optimizirana sekvenca instrukcija za oglednu arhitekturu koja se izvršava bez zastoja, izgleda ovako:

```
Loop: LD      F0, 0(R1)
      LD      F6, -8(R1)
      LD      F10, -16(R1)
      LD      F14, -24(R1)
      ADDD    F4, F0, F2
      ADDD    F8, F6, F2
      ADDD    F12, F10, F2
      ADDD    F16, F14, F2
      SD      0(R1), F4
      SD      -8(R1), F8
      SD      -16(R1), F12
      SUBI    R1, R1, #32
      BNEZ    R1, Loop
      SD      8(R1), F16        ; - 32 + 8 = -24 (u odnosu na (R1))
```

Ovo se izvrši za 14 ciklusa - 3.5 po kopiji originalne petlje, što pretstavlja značajno ubrzanje. "Odmotavanje" petlje povećava segmente koda bez grananja koji se mogu efikasno raspoređivati.

Ubrzanje zavisi od toga koliko su instrukcije međusobno (ne)zavisne. Nezavisne - paralelne instrukcije se mogu istovremeno izvršavati i mogu im se zamijeniti mjesta u programu.

Postavlja se pitanje kako odrediti zavisnosti?

Postoje tri tipa zavisnosti:

1. zavisnost od podataka (eng. data dependences),
2. zavisnost imena (eng. name dependences),
3. upravljačka zavisnost (eng. control dependences).

Instrukcija "j" je zavisna od podataka instrukcije "i" ako "i" proizvodi rezultat koji "j" koristi (direktno ili indirektno preko neke treće instrukcije, pa i kroz čitav program). Zavisnost od podataka je lako otkriti u registrima ali teško u memoriji.

Primjeri:

```
LD      F0, 0(R1)     ; element niza u F0
ADDD    F4, F0, F2    ; dodaj skalar iz F2, F0 izaziva zavisnost sa LD
SD      0(R1), F4     ; upis rezultata, F4 izaziva zavisnost sa ADDD
i
SUBI    R1, R1, #8    ; umanji pokazivač za 8 bajta
…
BNEZ    R1, Loop      ; grananje za R1!=0, R1 izaziva zavisnost sa SUBI
```

Zavisnosti diktiraju redoslijed koji se mora zadržati, inače nastaju RAW hazardi.

Zavisnosti su osobine programa, dok su eventualni hazardi osobine/posljedice organizacije PS-e. U gornjem primjeru LD - ADDD izaziva zastoj ali SUBI - BNEZ ne jer se može izbjeći proslijeđivanjem (osobina PS-e). Prisustvo zavisnosti ukazuje na mogućnost pojave hazarda, a da li će do njega doći i koliki će zastoj izazvati - zavisi od osobina PS-e.

Zavisnost imena se javlja kada dvije instrukcije koriste isti registar ili memorijsku lokaciju (ime) ali bez protoka podataka među njima. Ona odgovara WAR hazardu, kada se zove antizavisnost (eng. antidependence) ili WAW hazardu, kada se zove izlazna zavisnost (output dependence - kada "i" i "j" pišu po istom registru/memorijskoj lokaciji - kada se redoslijed mora zadržati). Instrukcije "i" i "j" se mogu izvršavati paralelno ili obrnutim redoslijedom ako se "ime" promijeni tako da se izbjegne konflikt. Kod registara je to jednostavno riješiti reimenovanjem registara (eng. register renaming) i može se izvoditi statički/kompajlerom ili dinamički/hardverom.

Primjer:

```
odmotana petlja bez …                    ...i sa reiminovanjem registara
Loop: LD   F0, 0(R1)                Loop: LD   F0,0(R1)
      ADDD F4, F0, F2                     ADDD F4, F0, F2
      SD   0(R1), F4    ;bez SUBI i BNEZ; SD   0(R1), F4
      LD   F0, -8(R1)                     LD   F6, -8(R1)
      ADDD F4, F0, F2                     ADDD F8, F6, F2
      SD   -8(R1), F4   ;bez SUBI i BNEZ; SD   -8(R1), F8
      LD   F0, -16(R1)                    LD   F10, -16(R1)
      ADDD F4, F0, F2                     ADDD F12, F10, F2
      SD   -16(R1), F4  ;bez SUBI i BNEZ; SD   -16(R1), F12
      LD   F0, -24(R1)                    LD   F14, -24(R1)
      ADDD F4, F0, F2                     ADDD F16, F14, F2
      SD   -24(R1), F4                    SD   -24(R1), F16
      SUBI R1, R1, #32                    SUBI R1, R1, #32
      BNEZ R1, Loop                       BNEZ R1, Loop
```

U lijevoj sekvenci instrukcija postoje i zavisnosti podataka (npr. kroz F0 u prvoj i drugoj instrukciji) i zavisnosti imena (F0 kod LD i ADDD instrukcija i F4 kod ADDD), dok u desnoj postoji samo zavisnost podataka (samo unutar tijela petlji - LD-ADDD sa F0 u prvoj i ADDD-SD sa F4 u prvoj). Nakon reimenovanja registara u svakoj kopiji tijela petlje, svako tijelo postaje nezavisno i može se izvršavati paralelno, uz preklapanje ili mimo reda.

Upravljačke zavisnosti određuju raspoređivanje instrukcija obzirom na instrukcije grananja tako da se instrukcija ne-grananja izvrši samo kada treba.

Primjer:

```
if p1 {
        S1;     /* S1 je upravljački zavisna od p1 */
};
if p2 {
        S2;     /* S2 je zavisna od p2 ali ne i od p1 */
}
```

Ograničenja, obzirom na upravljačke zavisnosti, se mogu podijeliti u dvije grupe:

1. upravljački zavisna instrukcija zavisna od grananja se ne može prebaciti prije grananja - da od njega ne zavisi, i
2. instrukcija koja nije zavisna od grananja, se ne može prebaciti iza grananja - da od njega zavisi.

Primjer uklanjanja upravljačke zavisnosti je dat na "odmotanoj" petlji sa zadržanim, ali modifikovanim, grananjima.

```
Loop: LD      F0, 0(R1)
      ADDD    F4, F0, F2
      SD      0(R1), F4
      SUBI    R1, R1, #8   ; bez BNEZ
      BEQZ    R1, Exit
      LD      F6, 0(R1)
      ADDD    F8, F6, F2
      SD      0(R1), F8
      SUBI    R1, R1, #8   ; bez BNEZ
      BEQZ    R1, Exit
      LD      F10, 0(R1)
      ADDD    F12, F10, F2
      SD      0(R1), F12
      SUBI    R1, R1, #8   ; bez BNEZ
      BEQZ    R1, Exit
      LD      F14, 0(R1)
      ADDD    F16, F14, F2
      SD      0(R1), F16
      SUBI    R1, R1, #8   ; bez BNEZ
      BNEZ    R1, Loop
Exit:
```

Zbog BEQZ grananja, upravljačke zavisnosti (sa LD, ADDD, SUBI i sljedeće BEQZ) sprječavaju preklapanja izvršenja instrukcija. Zbog BEQZ moraju postojati i SUBI instrukcije.

Ako je vrijednost u R1 multipl od 32 a broj ponavljanja petlji multipl od 4, može se zaključiti da se BEQZ instrukcije nikada neće izazvati grananje, pa nema upravljačke zavisnosti. Izbacivanje instrukcija grananja četiri SUBI instrukcija se mogu zamijeniti jednom - kao u

ranijim primjerima.

Primjer pokazuje da upravljačka zavisnost nije osobina koja se mora sačuvati. Ono što se mora sačuvati su:

1. **ponašanje izuzetaka** (eng. exception behavior) - mijenjanje redoslijeda izvršenja instrukcija ne smije promijeniti način pojave izuzetaka u programu, a pogotovo, ne smije uvesti nove.

Primjer: bez korištenja odgođenog grananja

```
        BEQZ   R2, L1
        LW     R1, 0(R2)
L1:
```

nema zavisnosti podataka. Ako se ignorišu upravljačke zavisnosti i zamijeni redoslijed instrukcija LW može izazvati izuzetak kod pristupa memoriji. Ako bi se grananje desilo, takav izuzetak se nebi desio!

2. **tok podataka** - kako grananja čine tok dinamičkim, podaci mogu dolaziti iz više izvora.

Primjer:

```
        ADD    R1, R2, R3
        BEQZ   R4, L
        SUB    R1, R5, R0
L:      OR                  R7, R1, R8
```

Vrijednost u R1, koju OR koristi, zavisi od ishoda grananja. Zavisnost podataka, sama za sebe, ne čuva korektnost koda jer se bavi samo statičkim rasporedom čitanja i pisanja - mora se sačuvati tok podataka: ako se ne desi grananje, vrijednost R1 izračunata u SUB ide OR instrukciji, a ako se desi, R1 je izračunat u ADD i ide u OR. Očuvanjem upravljačke zavisnosti SUB od grananja, sprječava se promjena toka podataka.

6.1. Paralelizam na nivou petlje

Paralelizam na nivou petlje (eng. Loop-Level Parallelism - LLP) je osobina koda koja omogućava se poslovi unutar petlje (tijelo petlje) izvršavaju uz preklapanje, paralelno ili mimo reda. On se obično analizira na nivou izvornog koda (ili blizu) dok se analiza ILP-a obavlja na nivou instrukcija koje generiše kompajler. Dio LLP analize je određivanje zavisnosti unutar i između iteracija petlje.

Primjer:

```
for (i=1; i<=1000; i++)
      x[i]=x[i]+s;
```

Ovdje postoji zavisnost u tijelu petlje (dva pristupa i-tom elementu niza x), ali ne i među instrukcijama u različitim iteracijama. Iz toga se zaključuje da je petlja paralelna. Prevodom u asembler, pravi se zavisnost koju nosi petlja (loop-carried dependence) kao što je R1 (adresa/dekrement) u ranijim primjerima.

Složeniji slučaj petlje je

```
for (i=1; i<=1000; i++) {
```

```
        A[i+1]=A[i]+C[i];       /* S1 */
        B[i+1]=B[i]+A[i+1];     /* S2 */
}
```

Ovdje postoje dvije različite zavisnosti:

1. S1 koristi vrijednost izračunatu u S1 u prethodnom prolazu, S2 isto radi sa B[i] i B[i+1],
2. S2 koristi A[i+1] izračunat u S1 u istom prolazu.

S1 zavisi od S1 u ranijem prolazu i to je zavisnost koju nosi petlja - zato se iteracije moraju obavljati redom. S2 zavisi od S1 unutar petlje i nema zavisnosti petlje. Ako su to jedine zavisnosti, više iteracija se može izvršiti paralelno sve dok su S1 i S2 u paru i u datom redoslijedu.

6.2. Dinamičko raspoređivanje instrukcija

U ranijim poglavljima je bilo riječi o načinima na koje kompajleri mogu raspoređivati instrukcije sa ciljem da razdvoje međusobno zavisne instrukcije te minimiziraju broj stvarnih hazarda i odgovarajućih zastoja. Takav pristup se naziva statičkim raspoređivanjem i uveden je 1960-tih a postao popularan 1980-tih kada su arhitekture sa protočnim strukturama postale široko rasprostranjene.

Kod dinamičkog raspoređivanja hardver raspoređuje izvršenje instrukcija sa ciljem smanjivanja broja zastoja, a time povećanja performansi procesora.

Ovaj pristup ima više prednosti:

1. omogućava rješavanje situacija kod kojih zavisnosti nisu poznate u trenutku kompajliranja (zbog npr. pristupa memoriji),
2. olakšavaju posao kompajlerima i čini ih jednostavnijim,
3. omogućava efikasno izvršavanje koda kompajliranog za jednu PS-u na drugoj PS-i.

Dinamičko raspoređivanje zahtijeva značajno povećanje složenosti hardvera. Iako procesor sa dinamičkim raspoređivanjem ne može ukloniti zavisnosti podataka, on pokušava izbjeći zastoje kada su zavisnosti prisutne. Za razliku od toga, statičko raspoređivanje pokušava izbjeći zastoje razdvajanjem zavisnih instrukcija tako da se hazardi ne javljaju. Statičko raspoređivanje se može koristiti u kodu namijenjenom izvršenju na procesoru sa PS-om sa dimaničkim raspoređivanjem.

Glavno ograničenje, do sada razmatranih tehnika PS-a, je to da one pokreću instrukcije redom - ako je jedna instrukcija u zastoju, nijedna instrukcija iza nje se ne pokreće (svake dvije bliske zavisne instrukcije izazivaju zastoj).

Primjer:

```
DIVD   F0, F2, F4
ADDD   F10, F0, F8
SUBD   F12, F8, F14
```

SUBD se ne može izvršiti zbog zavisnosti ADDD od DIVD i nastaje zastoj iako SUBD nije zavisna od podataka bilo koje instrukcije u PS-i.

U oglednoj protočnoj strukturi se i strukturni i hazardi podataka provjeravaju u ID fazi i tek kada se ustanovi da nema hazarda, instrukcija iza ID se pokreće dalje. Da bi bilo moguće

početi izvršenje SUBD iz gornjeg primjera, mora se proces pokretanja podijeliti na dva dijela - provjeru strukturnih hazarda i čekanje na prestanak hazarda podataka. Provjera na strukturne hazarde je moguća i nakon pokretanja instrukcije - tako se instrukcije pokreću redom, a potrebno je da izvršenje instrukcije počne čim njeni podaci postanu raspoloživi. Tako se u protočnoj strukturi izvršavaju instrukcije mimo reda, što dovodi i do njihovog završetka izvršenjamimo reda.

Izvršenje mimo reda pretstavlja veliki problem kod obrade prekida.

ID faza se sada dijeli na dva dijela:

1. Pokretanje - dekodiranje instrukcije i provjeru strukturnih hazarda i
2. Čitanje operanada - čekanje na prestanak hazarda podataka, čitanje operanada.

IF stepen može smještati instrukcije u jednostepeni ili višestepeni red čekanja, odakle ih proslijeđuje u stepen za pokretanje. EX stepen se nalazi iza stepena za čitanje operanada, kao kod ogledne arhitekture. Kao i kod FP PS-e, izvršenje može trajati više ciklusa. Zato se mora razlikovati kada instrukcija počinje izvršenje a kada završava izvršenje (između toga ona je "u izvršenju"). Tako se više instrukcija može naći "u izvršenju". Pored ovih promjena u PS-i, promijeniće se i dizajn funkcionalnih jedinica, njihov broj, kašnjenja koja unose, njihova protočnost - sve sa ciljem boljeg pretstavljanja naprednih mehanizama PS-e.

6.3. Dinamičko raspoređivanje instrukcija sa semaforom

Kod protočne strukture sa dinamičkim raspoređivanjem instrukcija, sve instrukcije prolaze kroz stepen za pokretanje po redu (donošenja), nakon čega mogu biti u zastoju ili se proslijeđivati u sljedećem stepenu (čitanja operanada) i ući u izvršenje mimo reda. Semaforiranje (eng. scoreboarding) je tehnika koja ovo omogućava kada postoji dovoljno resursa i ne postoje zavisnosti podataka. Prvi put je korištena u CDC 6600 superračunaru iz sredine 1960-ih godina.

WAR hazardi, koji nisu postojali u cjelobrojnoj i FP PS-i, mogu se javiti ako se instrukcije izvršavaju mimo reda.

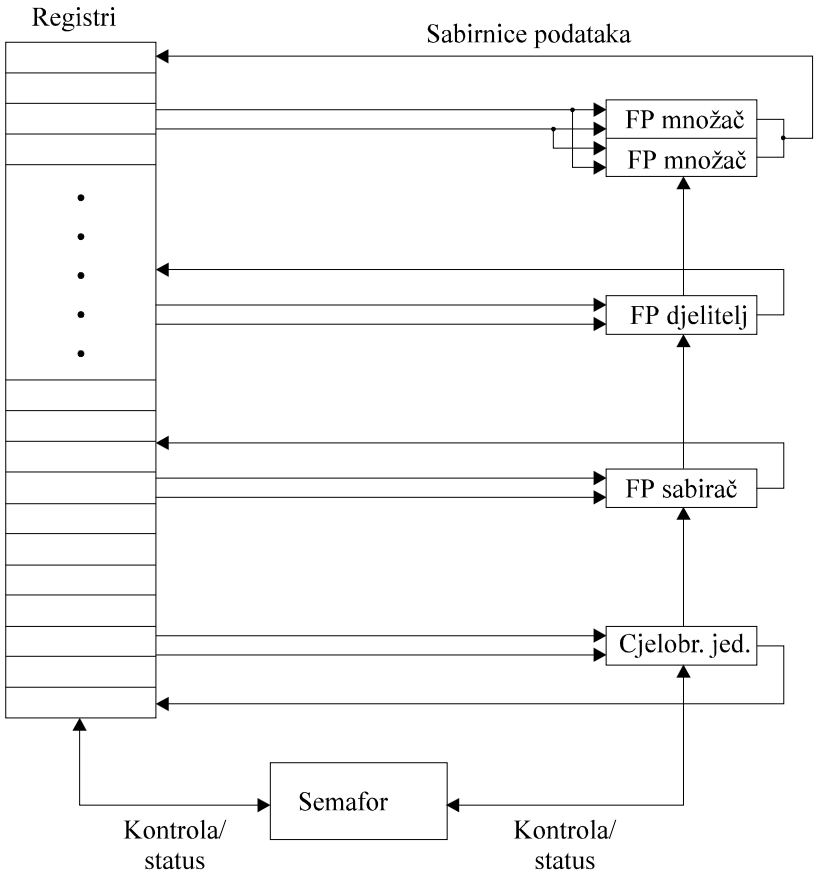
Primjer: neka je, u prethodnom primjeru F8 odredište SUBD instrukcije

```
DIVD   F0, F2, F4
ADDD   F10, F0, F8
SUBD   F8, F8, F14
```

Sada postoji antizavisnost između ADDD i SUBD, ako se u protočnoj strukturi izvrši SUBD prije ADDD. Slično, moguća je pojava izlazne zavisnosti (WAW - hazard) - kada bi odredište SUBD bio F10.

Cilj semaforisanja je da održi ritam izvršenja od jedne instrukcije po ciklusu sata (kada nema strukturnih hazarda) izvršavanjem instrukcije što prije je to moguće. Kada se instrukcija, koja je na redu za izvršavanje, nađe u zastoju, mogu se pokrenuti i izvršiti neke druge, koje ne zavise od jedne instrukcije u izvršenju (aktivne) ili u zastoju. Da bi se iskoristile prednosti izvršavanja mimo reda, potrebno je da više instrukcija bude u EX fazi istovremeno. To se može postići sa više funkcionalnih jedinica ili sa protočnim FJ-ma, (ili oboje) što je, sa stanovišta upravljanja ekvivalentno.

Kod ogledne arhitekture semaforisanje ima smisla prvenstveno kod FP-jedinice, jer su kašnjenja ostalih mala. Neka postoje dva množača, jedan sabirač, jedan djelitelj i jedna cjelobrojna jedinica za sve pristupe memoriji, grananja i cjelobrojne operacije. Ovakva konfiguracija sa semaforom je prikazana na slici 6.1..



Slika 6.1. Osnovna struktura ogledne arhitekture sa semaforom.

Svaka instrukcija ide kroz semafor, gdje se zapisuju zavisnosti podataka. Taj korak odgovara pokretanju instrukcije i zamjenjuje dio ID faze kod ogledne PS-e. Semafor zatim određuje (čeka i nadgleda promjene) kada instrukcija može pročitati svoje operande i početi izvršenje. On također kontroliše kada instrukcija može upisati svoj rezultat u odredišni registar.

Svaka instrukcija se izvršava u četiri koraka (FP-opracije ne rade pristup memoriji) koji zamjenjuju ID, EX i WB korake kod ogledne PS-e.

1. **Izdavanje** - Ako je FJ za instrukciju slobodna (neće doći do strukturnog hazarda) i nijedna aktivna instrukcija nema isti odredišni registar (neće doći do WAW-hazarda), semafor izdaje/pokreće izvršenje instrukcije na FJ-i i osvježava svoje unutrašnje strukture podataka. U protivnom instrukcija izaziva zastoj u izdavanju i puni bafer (latch ili red čekanja) između IF-a i izdavanja.
2. **Čitanje operanada** - Semafor nadgleda raspoloživost izvornih operanada (nijedna ranije pokrenuta instrukcija ili aktivna FJ neće pisati po njemu). Ako su oni raspoloživi, semafor kaže FJ-i da nastavi sa čitanjem i počne izvršavanje. RAW-hazardi se dinamički rješavaju i omogućava se pokretanje izvršenja mimo reda. Ovaj, kao i prethodni korak, obavljaju ulogu ID kod ogledne PS-e.
3. **Izvršenje** - FJ počinje izvršenje čim pročita izvorne operande. Kada završi,

obavještava semefor o tome (kao EX u oglednoj PS-i).

4. **Pisanje rezultata** - Semafor provjerava WAR-hazarde i izaziva zastoje ako je potrebno.

Primjer:

```
DIVD   F0, F2, F4
ADDD   F10, F0, F8
SUBD   F8, F8, F14
```

ADDD ima izvorni operand u F8, koji je odredište SUBD. Istovremeno, ADDD zavisi od prethodne instrukcije. Semafor mora zaustaviti SUBD u njenoj fazi pisanja rezultata dok ADDD ne pročita svoj operand. Ako ne postoji WAR-hazard, ili je već riješen, semafor kaže FJ-i da izvrši upis rezultata (kao WB u oglednoj PS-i).

Operandi se čitaju tek kada su oba raspoloživa - semafor ne koristi prosljeđivanje! Ali zato, za razliku od ogledne PS-e, rezultati se upisuju čim su spremni (ako nema WAR-hazarda) ne čekajući da dođu u segment za to. Tako se skraćuje kašnjenje PS-e i nadoknađuje propušteno kod prosljeđivanja. Dodatno kašnjenje se javlja jer se koraci 2 i 4 ne mogu preklapati (trebalo bi dodatno baferovanje).

Na osnovu svojih struktura podataka, semafor upravlja napredovanjem instrukcije od jednog koraka do drugog, komunicirajući sa FJ-ma.

Tabela 6.2. prikazuje izgled informacija semafora u jednom trenutku izvršenja sljedeće sekvence instrukcija:

```
LD     F6, 34(R2)
LD     F2, 45(R3)
MULD   F0, F2, F4
SUBD   F8, F6, F2
DIVD   F10, F0, F6
ADDD   F6, F8, F2
```

Postoje tri dijela semafora:

1. Status instrukcije - pokazuje u kojem od četiri koraka je instrukcija,
2. Status funkcionalne jedinice - pokazuje status svake FJ na po 8 polja:
   - Busy - govori da li je FJ zauzeta ili nije,
   - Op - daje operaciju koju FJ treba da izvrši,
   - Fi - odredišni registar,
   - Fj, Fk - brojevi izvorišnih registara,
   - Qj, Qk - FJ-e koje proizvode podatike u Fj i Fk,
   - Rj, Rk - flegovi koji govore kada su Fj i Fk spremni.
3. Status registara rezultata - pokazuje koja FJ će pisati u koji registar, ako trenutno aktivna instrukcija ima taj registar za odredište. Ovo polje je prazno kada ne postoji instrukcija koja će pisati po tom registru.

	Status instrukcija			
Instrukcija	Pokretanje	Čitanje operanada	Izvršenje završeno	Upis rezultata
LD F6, 34(R2)	√	√	√	√
LD F2, 45(R3)	√	√	√	
MULTD F0, F2, F4	√			
SUBD F8, F6, F2	√			

DIVD F10, F0, F6	√
ADDD F6, F8, F2	

	Status funkctionalnih jedinica								
Ime	Zauzeto	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Cjelobr.	Da	Load	F2	R3				Ne	
Množač1	Da	Mult	F0	F2	F4	Cjelobr.		Ne	Da
Množač2	Ne								
Sabirač	Da	Sub	F8	F6	F2		Cjelobr.	Da	Ne
Djelitelj	Da	Div	F10	F0	F6	Množ1		Ne	Da

	Status rezultata u registrima								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FJ	Mult1	Cjelobr.			Sub	Djelitelj			

Tabela 6.2. Komponente semafora. Svaka instrukcija koja je pokrenuta ili čeka na pokretanje ima svoje mjesto u tabeli statusa instrukcija.

Tabela 6.2. pokazuje da je prva LD završena i upisala je rezultat, druga LD je završila ali nije upisala rezultat. MULTD, SUBD i DIVD su pokrenute ali su u zastoju jer čekaju operande. Status FJ kaže da prvi množač čeka cjelobrojnu jedinicu, sabirač čeka cjelobrojnu jedinicu, a djelitelj čeka prvog množača. ADDD je u zastoju zbog strukturnog hazarda, koji će biti razriješen kad se završi SUBD.

Nastavak izvršenja je dat na tabeli 6.3. (semafor neposredno prije upisa razultata MULTD) i slici 6.4. (semafor neposredno prije upisa rezultata DIVD).

	Status instrukcija			
Instrukcija	Pokretanje	Čitanje operanada	Izvršenje završeno	Upis rezultata
LD F6, 34(R2)	√	√	√	√
LD F2, 45(R3)	√	√	√	√
MULTD F0, F2, F4	√	√	√	
SUBD F8, F6, F2	√	√	√	√
DIVD F10, F0, F6	√			
ADDD F6, F8, F2	√	√	√	

	Status funkctionalnih jedinica								
Ime	Zauzeto	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Cjelobr.	Ne								
Množač1	Da	Mult	F0	F2	F4			Ne	Ne
Množač2	Ne								
Sabirač	Da	Add	F6	F8	F2			Ne	Ne
Djelitelj	Da	Div	F10	F0	F6	Množ1		Ne	Da

	Status rezultata u registrima								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FJ	Množ1			Sabirač		Djelitelj			

Tabela 6.3. Tabele semafora neposredno prije prelaska MULTD u fazu pisanja rezultata. DIVD još nije pročitala nijedan od svojih operanada, jer ima zavisnost od rezultata množenja. ADDD je pročitala svoje operande i izvršava se, iako je bila prisiljena čekati dok SUBD ne završi, da bi se oslobodila funkcionalna jedinica. ADDD ne može nastaviti sa upisom rezultata zbog WAR hazarda u F6, koji koristi DIVD.

	Status instrukcija			
Instrukcija	Pokretanje	Čitanje operanada	Izvršenje završeno	Upis rezultata

LD F6, 34(R2)	√	√	√	√
LD F2, 45(R3)	√	√	√	√
MULTD F0, F2, F4	√	√	√	√
SUBD F8, F6, F2	√	√	√	√
DIVD F10, F0, F6	√	√	√	
ADDD F6, F8, F2	√	√	√	√

	Status funkctionalnih jedinica								
Ime	Zauzeto	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Cjelobr.	Ne								
Množač1	Ne								
Množač2	Ne								
Sabirač	Ne								
Djelitelj	Da	Div	F10	F0	F6			Ne	Ne

	Status rezultata u registrima								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FJ						Djelitelj			

Tabela 6.4. Tabele semafora neposredno prije prelaska DIVD u fazu pisanja rezultata. ADDD je bila u situaciji da se završi čim je DIVD prošla čitanje operanada i dobila kopiju F6. Ostalo je samo da se završi DIVD.

Tabela 6.5. prikazuje šta je potrebno semaforu za napredovanje i "knjigovodstvo" potrebno kada instrukcija napreduje. Imena registara pod navodnicima pretstavljaju imena a ne sadržaje istih. D je ime odredišnog registra, S1 i S2 su imena izvorišnih registara a op je operacija koju treba izvršiti.

Status instrukcije	Čekanje na	Knjigovodstvo
Pokretanje	Nije zauzeta (FJ) i ne rezultat (D)	Busy(FJ)←yes; Op(FJ)←op; Fi(FJ)←'D'; Fj(FJ)←'S1'; Fk(FJ)←'S2'; Qj←Result('S1'); Qk←Result('S2'); Rj←not Qj; Rk←not Qk; Result('D')← FJ;
Čitanje operanada	Rj i Rk	Rj←No; Rk←No
Izvršenje završeno	Funkcionalna jedinica završila	
Upis rezultata	∀ *f*((Fj(*f*)≠Fi(FJ) or Rj(*f*)=No) & (Fk(*f*)≠Fi(FJ) or Rk(*f*)=No))	∀ *f*(if Qj(*f*)= FJ then Rj(*f*)←Yes); ∀ *f*(if Qk(*f*)= FJ then Rk(*f*)←Yes); Result(Fi(FJ))←0; Busy(FJ)←No

Tabela 6.5. Neophodne provjere i knjigovodstvo za svaki korak u izvršenju instrukcije. FJ je finkcionalna jedinica koju instrukcija koristi, D je ime odredišnog registra, S1 i S2 su imena izvorišnih registara, i op je operacija koju treba odraditi. Za pristup polju Fj semafora, za funkcionalnu jedinicu FJ koristi se notacija Fj(FJ). Result(D) je vrijednost polja registra rezultata za registar D. Test u slučaju upisa rezultata sprječava upis kada postoji WAR hazard, koji nastaje ako druga instrukcija ima odredište ove instrukcije (Fi(FJ)) kao izvorište (bilo Fj(f) ili Fk(f)), i ako neka druga instrukcija pisala u registar (Rj=Yes ili Rk=Yes). Oznaka 'Ri' se koristi za ime registra Ri, a ne za njegov sadržaj.

Semafor koristi raspoloživi ILP da minimizira broj zastoja zbog zavisnosti podataka u programu. Njegova ograničenja su:

1. Nivo raspoloživog paralelizma među instrukcijama,
2. Broj podataka u semaforu - određuje koliko unaprijed PS-a može tražiti nezavisne instrukcije (za sada taj "prozor" se ne prostire dalje od eventualnog grananja),
3. Broj i tip funkcionalnih jedinica - određuje važnost strukturnih hazarda, može porasti kada se koristi dinamičko raspoređivanje instrukcija, i

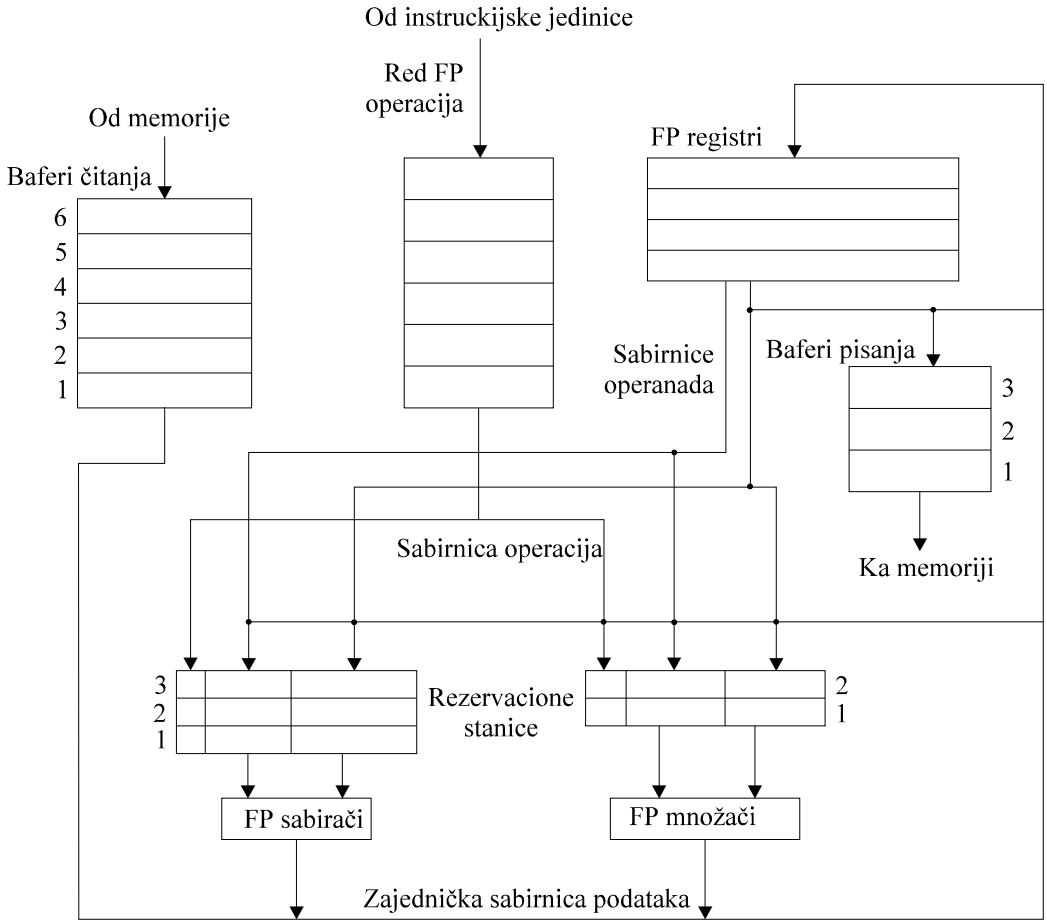
4. Prisustvo antizavisnosti i izlaznih zavisnosti (WAR i WAW zastoja).

6.4. Dinamičko raspoređivanje instrukcija pomoću Tomasulovog algoritma

Ovaj pristup dinamičkom raspoređivanju instrukcija u prisustvu hazarda je dobio ime po Robertu Tomasulo-u i prvi put je realizovana u FP-jedinici računara IBM 360/91, tri godine nakon pojave CDC 6600, a prije pojave keš-memorija. U njemu se kombinuju osnovni elementi semaforiranja i uvođenje reimenovanja registara (u cilju izbjegavanja WAR i WAW hazarda). Ulogu kompajlera pri reimenovanju registara i izbjegavanju hazarda, u ovom slučaju preuzimaju tzv. rezervacione stanice (RS, eng. reservation stations), koje baferuju operande za instrukcije koje čekaju da budu pokrenute, i logika za pokretanje tih instrukcija. Osnovna ideja je u tome da rezervaciona stanica pribavlja i baferuje operand čim on postane raspoloživ, eliminišući potrebu da se on uzima iz registra. Upravljačka logika o tome "obavještava" instrukciju "na čekanju" tako da, kada se ona pokrene, polja specifikatora registara promijeni (reimenovanje registara) u ime rezervacione stanice. Ova dodatna mogućnost je glavna razlika između semaforiranja i Tomasulovog algoritma. Kada se javi više uzastopnih upisa u isti registar, samo zadnji se koristi za stvarno osvježavanje stanja registra. Kako rezervacionih stanica može biti više od registara, ovako se mogu eliminisati hazardi koji se ne mogu eliminisati pomoću kompajlera.

Pored reimenovanja registara, postoje još dvije razlike između Tomasulovog algoritma i semaforiranja. Prvo, otkrivanje hazarda i upravljanje izvršenjem je distribuirano - rezervacione stanice kod svake FJ-e određuju kada može početi izvršavanje instrukcije na toj FJ-i (što je centralizovano kod semaforiranja). Drugo, rezultati se prosljeđuju direktno iz rezervacionih stanica, gdje su baferovani, do FJ-a kojima su potrebni (a ne kroz registre). To se radi pomoću zajedničke sabirnice podataka (CDB od eng. Common data bus), koja omogućava svim jedinicama koje čekaju rezultat da ga dobiju istovremeno (kod semaforiranja to ide kroz registre, pa se onda FJ-e za njih "bore").

Slika 6.2. daje osnovnu strukturu FP-jedinice ogledne arhitekture bazirane na Tomasulovom algoritmu. Nije data nijedna tabela za upravljanje izvršenjem. Rezervacione stanice sadrže instrukcije koje su već bile pokrenute i čekaju na izvršenje na FJ-i, operande za tu instrukciju (ako su već izračunati) ili izvor operanada, kao i podatke neophodne za upravljanje u toku izvršenja na FJ-i. Ulazni (load) i izlazni (store) baferi sadrže podatke ili adrese koji stižu od, ili odlaze ka memoriji. FP registri su povezani dvjema sabirnicama sa FJ-ama i jednom sa izlaznim (store) baferima. Svi rezultati iz FJ-a i memorije se šalju na zajedničku sabirnicu podataka, kojom su vezani za sva odredišta osim ulaznih (load) bafera. Svi baferi i rezervacione stanice imaju tag-polja koja se koriste za kontrolu hazarda.



Slika 6.2. Osnovna struktura FP jedinice ogledne arhitekture sa Tomasulo-vim algoritmom. Kada se pokrenu, FP operacije se šalju iz instrukcijske jedinice u red čekanja. U rezervacionim stanicama su operacije i operandi, kao i informacije potrebne za otkrivanje i rješavanje hazarda. Baferi čitanja sadrže rezultate tekućih čitanja iz memorije, a baferi pisanja sadrže adrese tekućih upisa u memoriju koji čekaju na svoje operande. Svi rezultati iz FP jedinica ili jedinice čitanja iz memorije se šalju na zajedničku sabirnicu podataka (CDB), koja vodi do FP registara, kao i rezervacionih stanica i bafera upisa u memoriju. FP sabirači odrađuju sabiranje i oduzimanje, dok FP množači rade množenje i dijeljenje.

Pošto se operandi prenose drugačije nego kod semafora, postoje samo tri koraka/faze kroz koje prolazi instrukcija:

1. **Pokretanje** - uzima instrukciju iz reda FP-instrukcija. Ako je uzeta FP-instrukcija, pokreće se ako postoji slobodna/prazna rezervaciona stanica, i, ako su operandi u registrima, šalju se u tu rezervacionu stanicu. Ako je uzeta instrukcija čitanje (load) ili pisanja (store), pokreće se ako postoji prazan odgovarajući bafer. Ako ne postoji prazna RS ili prazan bafer, postoji strukturni hazard i instrukcija se zadržava dok se RS ili bafer ne oslobodi. Ovaj korak vrši i reimenovanje registara.
2. **Izvršenje** - Ako jedan ili više operanada još nisu raspoloživi, prati se CDB dok se čeka da se izračuna sadržaj registra. Kada je operand raspoloživ, šalje se u odgovarajuću RS-u. Kada su oba raspoloživa izvrši se operacija. Ovaj korak provjerava RAW hazarde.
3. **Pisanje rezultata** - kada je rezultat raspoloživ, piše se na CDB, a odatle u registre i FJ-e koje taj rezultat očekuju.

Iako su ovi koraci slični onim kod semafora, postoje tri važne razlike. Prvo, ne postoje provjere na WAW i WAR hazarde - oni su eliminisani prilikom reimenovanja registara u fazi

pokretanja. Drugo, CDB služi za “emitovanje” rezultata, umjesto slanja kroz registre. Treće, ulazni (load) i izlazni (store) baferi se tretiraju kao osnovne FJ-e.

Određene strukture podataka za otkrivanje i eliminisanje hazarda su pridodate RS-ama, skupu registara i ulaznim i izlaznim baferima. Osim kod ulaznih bafera sve strukture imaju tag-polja. U njima su imena proširenog skupa virtuelnih registara koji se koriste prilikom reimenovanja. Na primjer, 4-bitno polje za označavanje jedne od 5 RS-a ili jednog od 6 ulaznih bafera. Tag-polje opisuje koja RS-a sadrži instrukciju koja će dati rezultat potreban kao izvorni operand. Kada se instrukcija pokrene i čeka na rezultat, ona pristupa tom operandu preko broja RS-e, a ne preko broja odredišnog registra upisanog od strane instrukcije koja proizvodi tu vrijednost. Kako postoji više RS-a nego stvarnih registara, WAW i WAR hazardi se mogu smanjiti reimenovanjem rezultata pomoću broja RS. Kod Tomasulovog algoritma RS-e se koriste kao prošireni virtuelni registri.

Važno je napomenuti da se tag-polja odnose na bafer ili FJ koja će proizvesti rezultat. Imena registara se odbacuju kada je instrukcija pokrenuta u RS-i.

Svaka RS ima 6 polja:
Op - operacija koju treba izvršiti nad operandima S1 i S2,
Qj, Qk - RS-e koje će proizvesti odgovarajuće izvorne operande,
Vj, Vk - vrijednost izvornih operanada (samo jedno Q ili V polje je validno za svaki operand),
Busy - indicira da su RS i odgovarajuća FJ zauzeti.

Svaki registar iz skupa registara i izlazni bafer imaju Qi polje:

Qi - broj RS koja sadrži operaciju čiji rezultat treba biti smješten u ovaj registar ili memoriju.

Svaki od ulaznih i izlaznih bafera zahtjeva busy polje, pokazujući kada je bafer raspoloživ zbog završetka ulaza ili izlaza njima dodijeljenih. Izlazni bafer ima i polje V koje sadrži vrijednost koju treba upisati u memoriju.

Primjer izgleda tabela informacija za datu sekvencu instrukcija je dat na tabeli 6.6. - stanje kada je samo prva instrukcija (LD) upisala rezultat. Slika otkriva RS-e, ulazne i izlazne bafere i tag-polja registara. Brojevi dodati imenima add, mult i load predstavljaju tag za tu RS-u - Sabir1 je tag za rezultat iz prve FJ-e za sabiranje. Tabela statusa instrukcija nije dio hardvera već služi samo za razumijevanje algoritma.

U odnosu na semafor, postoje dvije očigledne razlike. Prvo, vrijednost operanda je smještena u RS-i u jednom od V polja, čim je on raspoloživ. Ona se ne čita iz skupa registara, niti iz neke druge RS-e, kada se instrukcija jadnom pokrene. Drugo, ADDD instrukcija koja je blokirana u semaforu WAR hazardom u WB fazi, je pokrenuta i mogla bi se završiti prije započinjanja DIVD.

	Status instrukcije		
Instrukcija	Pokretanje	Izvršenje	Upis rezultata
LD F6, 34(R2)	√	√	√
LD F2, 45(R3)	√	√	
MULTD F0, F2, F4	√		
SUBD F8, F6, F2	√		
DIVD F10, F0, F6	√		

ADDD F6, F8, F2	√

	Rezervaciona stanica					
Ime	Zauzeto	Op	Vj	Vk	Qj	Qk
Sabir1	Da	SUB	Mem[34+Regs[R2]]			Load2
Sabir2	Da	ADD			Sabir1	Load2
Sabir3	Ne					
Množ1	Da	MULT		Regs[F4]	Load2	
Množ2	Da	DIV		Mem[34+Regs[R2]]	Množ1	

	Status registara								
Polje	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Množ1	Load2		Sabir2	Sabir1	Množ2			

Tabela 6.6. Rezervacione stanice i oznake registara. Sve su instrukcije pokrenute, ali je samo prva instrukcija čitanja završena i upisala svoj rezultat na CDB. Tabela statusa instrukcija u stvarnosti ne postoji već se ekvivalentne informacije distribuiraju kroz hardver. Vj i Vk polja pokazuju vrijednosti operanda. Baferi čitanja i pisanja nisu prikazani. Load2 bafer je jedini zauzeti i radi za drugu instrukciju u sekvenci - čitanje sa memorijske adrese R3+45. Operand je, u svakom trenutku, određen ili Q ili V poljem.

Glavne prednosti Tomasulove šeme su:

1 distribucija logike za otkrivanje hazarda, i
2 eliminisanje zastoja zbog WAW i WAR hazarda.

Prva prednost proističe iz distribuiranih RS-a i upotrebe CDB-a. Ako više instrukcija čeka na jedan rezultat, i svaka instrukcija već ima drugi operand, sve te instrukcije se mogu pokrenuti istovremeno nakon "emitovanja" rezultata preko CDB-a. Kod semafora instrukcije moraju čitati rezultat iz registara kada su registarske sabirnice slobodne.

WAW i WAR hazardi su eliminisani reimenovanjem registara korištenjem RS-a, i procesom smještanja operanada u odgovarajuće RS-e, čim su oni raspoloživi.

Primjer na tabeli 6.6. - pokrenute su DIVD i ADDD iako postoji WAR hazard sa F6. Taj hazard se eliminiše na jedan od dva načina

– ako je instrukcija koja daje vrijednost za DIVD završena, tada će Vk smjestiti rezultat, omogućavajući izvršenje DIVD neovisno o ADDD (kao što je prikazano) i

– ako se LD nije završila, tada bi Qk pokazivala na Load1 RS-u i DIVD bi bila nezavisna od ADDD.

U oba slučaja, ADDD se može pokrenuti i početi izvršavanje. Svaka upotreba rezultata MULD bi pokazivala na RS, omogućavajući da se ADDD završi i smjesti svoj rezultat u registre bez uticaja na DIVD.

Primjer: neka su kašnjenja - sabiranje 2 ciklusa sata, množenje 10 i dijeljenje 40. Tabela 6.7. daje tri tabele kada je MULD spremna da piše svoj rezultat.

Za razliku primjera sa semaforom, ADDD se završila pošto su operandi od DIVD kopirani, sprječavajući WAR hazard. Čak iako je punjenje u F6 odgođeno, sabiranje u F6 se može izvršiti bez WAW hazarda.

	Status instrukcija		
Instrukcija	Pokretanje	Izvršenje	Upis rezultata
LD F6, 34(R2)	√	√	√
LD F2, 45(R3)	√	√	√

MULTD F0, F2, F4	√	√	
SUBD F8, F6, F2	√	√	√
DIVD F10, F0, F6	√		
ADDD F6, F8, F2	√	√	√

	Reservacione stanice					
Ime	Busy	Op	Vj	Vk	Qj	Qk
Sabir1	Ne					
Sabir2	Ne					
Sabir3	Ne					
Množ1	Da	MULT	Mem[45+Regs[R3]]	Regs[F4]		
Množ2	Da	DIV		Mem[34+Regs[R2]]	Mult1	

	Status registara								
Polje	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Množ1					Množ2			

Tabela 6.7. Množenje i dijeljenje su jedine nedovršene instrukcije. Ovo je različito od slučaja sa semaforom, jer eliminacija WAR hazardâ je omogućila završetak ADDD odmah nakon SUBD od koje je zavisila.

Tabela 6.8. daje korake kroz koje mora proći svaka instrukcija. Čitanje i pisanje su malo drugačije. Instrukcija čitanja se može izvršavati čim je raspoloživa. Kada je izvršavanje završeno, a CDB je slobodna, instrukcija čitanja emituje rezultat kao i svaka FJ-a. Instrukcije pisanja primaju svoje vrijednosti sa CDB ili iz skupa registara i izvršavaju se autonomno. Kada završe, isključuju busy polje da bi objavili raspoloživost, isto kao ulazni bafer ili RS.

Status instrukcije	Čekanje na	Posao knjigovodstva

Pokretanje	Stanica ili bafer prazan	if (Register['S1'].Qi≠0)
		{RS[r].Qj←Register['S1'].Qi}
		else {RS[r].Vj←S1; RS[r].Qj←0};
		if (Register[S2].Qi≠0)
		{RS[r].Qk←Register[S2].Qi};
		else {RS[r].Vk←S2; RS[r].Qk←0}
		RS[r].Busy←yes;
		Register['D'].Qi=r;

Izvršenje	(RS[r].Qj=0) and (RS[r].Qk=0)	Ništa - operandi su u Vj i Vk
Upis rezultata	Izvršenje završeno u r i CDB slobodan	∀x (if (Register[x].Qi=r) {Fx←result; Register[x].Qi←0}); ∀x (if (RS[x].Qj=r) {RS[x].Vj←result; RS[x].Qj←0}); ∀x (if (RS[x].Qk=r) {RS[x].Vk←result; RS[x].Qk←0}); ∀x (if (Store[x].Qi=r) {Store[x].V←result; Store[x].Qi←0}); RS[r].Busy←No

Tabela 6.8. Koraci u algoritmu i šta je potrebno za svaki od njih. Za instrukciju koja se pokreće, D je odredište, S1 i S2 su brojevi izvorišnih registara, i r je rezervaciona stanica ili bafer kome je D dodijeljeno. RS je struktura podataka rezervacione stanice. Vrijednost koju vraća rezervaciona stanica ili jedinice čitanja se zove rezultat. Register je registarska struktura podataka (ne skup registara), dok je Store struktura podataka bafera upisa. 'Ri' je ime registra Ri, a ne njegov sadržaj. Kada je instrukcija pokrenuta, odredišni registar ima svoje Qi polje postavljeno na broj bafera ili rezervacione stanice kojoj je instrukcija dodijeljena. Ako su operandi raspoloživi u registrima, oni su upisani u V polja. U suprotnom, Q polja indiciraju rezervacionu stanicu koja će proizvesti izvorišne operande. Instrukcija čeka u RS dok oba njena operanda ne postanu raspoloživi, što je označeno nulom u Q poljima. Q polja su u nuli kada je instrukcija pokrenuta, ili kada se instrukcija od koje ova instrukcija zavisi završi i upiše svoj rezultat. Kada se instrukcija izvrši i CDB je raspoloživa, ona može odraditi svoj upis. Svi baferi, registri i rezervacione stanice čija vrijednost Qj ili Qk je ista kao kod rezervacione stanice koja završava posao, uzima vrijednosti sa CDB i ažurira Q polja tako da pokazuju da su vrijednosti primljene.

Na taj način, CDB može poslati rezultat mnogim odredištima u jednom ciklusu sata, i ako instrukcije na čekanju imaju svoje operande, sve mogu početi izvršenje u sljedećem ciklusu sata.

Kao demonstraciju eliminisanja WAW i WAR hazarda dinamičkim reimenovanjem registara, poslužiće sljedeća petlja - množenje članova niza skalarom u F2:

```
Loop:   LD      F0, 0(R1)
        MULTD   F4, F0, F2
        SD      0(R1), F4
        SUBI    R1, R1, #8
        BNEZ    R1, Loop        ; grana se ako R1 nije 0
```

Ako se predvidi da će se grananja desiti, korištenje RS-a će omogućiti da više izvršenja ove petlje počne odmah, bez odmotavanja petlje - u stvari, petlja se odmotava dinamički (hardverom). Tomasulov algoritam podržava paralelno (preklapanjem) izvršavanje više kopija iste petlje, uz korištenje malog broja registara. RS-e proširuju realni skup registara pomoću reimenovanja. Ova tehnika je korištena samo u jačim (i skupljim) članovima familije procesora IBM360, u kojoj su svi modeli imali isti programski model (pa i skup registara) sa ciljem održanja binarna kompatibilnosti i prenosivosti softvera.

Neka su pokrenute sve instrukcije iz dvije uzastopne iteracije petlje, ali nijedna FP-load/store ili operacija nije završena. Stanje RS-a, tabela statusa registara i ulaznih i izlaznih bafera u tom trenutku su dati na tabeli 6.9. (cjelobrojna ALU operacija je ignorisana, i uzeto je da je predviđeno da će se desiti grananja). Tada su dvije kopije u izvršenju uz CPI blizu 1.0, uz uslov da se množenja mogu završiti za 4 ciklusa sata. Ako se ignoriše instrukcije za održavanje petlje (posmatra samo tijelo petlje), postignute performanse odgovaraju onima kod kompajlerski odmotane petlje i raspoređenih instrukcija, pod uslovom da je na raspolaganju dovoljno registara.

Load instrukcija iz druge iteracije petlje se može završiti prije *store*-a iz prve - što je u redu ako pristupaju različitim adresama. To se provjerava ispitivanjem adresa u izlaznom baferu kad god se pokreće load. Ako su adrese u ulaznim i izlaznim baferima iste - mora doći do zastoja dok izlazni bafer ne dobije svoju vrijednost. Tada mu se može pristupiti ili uzeti podatak iz memorije. To je alternativa kompajleru i njegovoj tehnici zamjene *load*-a i *store*-a.

	Status instrukcije			
Instrukcija	Iz iteracije	Pokretanje	Izvršenje	Upis rezultata
LD F0, 0(R1)	1	√	√	
MULTD F4, F0, F2	1	√		
SD 0(R1), F4	1	√		
LD F0, 0(R1)	2	√	√	
MULTD F4, F0, F2	2	√		
SD 0(R1), F4	2	√		

	Rezervacione stanice					
Ime	Zauzeto	Fm	Vj	Vk	Qj	Qk
Sabir1	Ne					
Sabir2	Ne					
Sabir3	Ne					
Množ1	Da	MULT		Regs[F2]	Load1	
Množ2	Da	MULT		Regs[F2]	Load2	

	Status registara								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Load2		Množ2						

	Baferi čitanja		
Polje	Load 1	Load 2	Load 3
Adresa	Regs[R1]	Regs[R1]-8	
Zauzeto	Da	Da	Ne

	Baferi pisanja		
Polje	Store 1	Store 2	Store 3
Qi	Množ1	Množ2	
Zauzeto	Da	Da	Ne
Adresa	Regs[R1]	Regs[R1]-8	

Tabela 6.9. Dvije aktivne iteracije bez ijedne završene instrukcije. Baferi čitanja i pisanja su dati sa adresama iz kojih treba čitati i u koje treba upisati. Pročitani podaci su u baferu čitanja; polja u rezervacionoj stanici množača pokazuju da su podaci iz bafera čitanja - izvorišta. Baferi upisa indiciraju da je odredište množača njihova vrijednost za upis.

Glavni nedostatak Tomasulovog algoritma je njegova složenost i to što zahtijeva mnogo hardvera. On kombinuje dvije tehnike: reimenovanje registara u prošireni virtuelni skup registara i baferovanje izvornih operanada iz registara (rješava WAR hazarde). Pogodan je u situacijama kada je teško rasporediti instrukcije za neku PS-u, ili kada postoji nedostatak raspoloživih registara.

7. Dinamičko predviđanje grananja

Učestalost grananja i skokova zahtijeva posebne tehnike za smanjenje zastoja zbog upravljačkih zavisnosti. Do sada je bilo govora o statičkim metodama rješavanja predviđanja grananja, koje ne uzimaju u obzir njegovo dinamičko ponašanje u toku izvršavanja programa.

Cilj svih mehanizama predviđanja grananja je da omoguće procesoru rano rješavanje problema grananja, sprječavajući da kontrolne zavisnosti izazovu zastoje. Učinak mehanizma predviđanja ne zavisi samo od njegove tačnosti, već i od cijena koje se plaćaju (kašnjenja) kod tačnog i netačnog predviđanja. Ovi parametri zavise od strukture PS-e, načina predviđanja i strategija koje se koriste za oporavak nakon pogrešnog predviđanja.

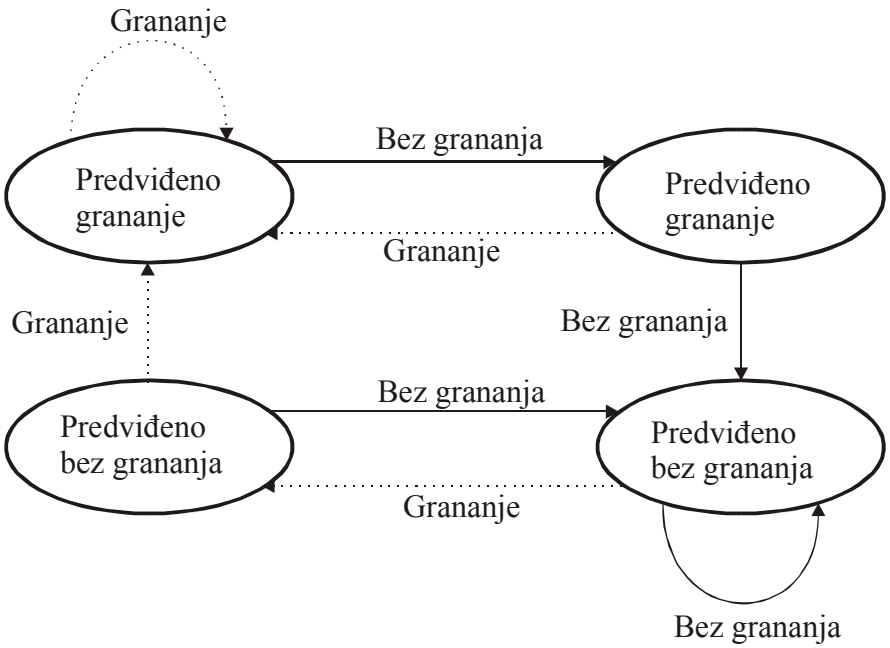
Najjednostavniji način dinamičkog (hardverskog) predviđanja grananja je pomoću bafera za predviđanje grananja (BPB, od eng. branch-prediction buffer) ili tabele istorije grananja (BHT, od eng. branch history table). Bafer za predviđanje grananja je mala memorija indeksirana donjim dijelom adrese instrukcije grananja. Ta memorija sadrži bit koji kaže da li se grananje desilo ili ne u zadnjem prolazu. Bafer nema tag-ova i koristan je samo za smanjenje kašnjenja kod grananja kada je ono veće od vremena potrebnog za izračunavanje moguće adrese grananja. Ne zna se da li je predviđanje ispravno - bit može postaviti i neko drugo grananje sa istim donjim bitima adrese. Uzima se kao nagovještaj da je tačno, i dobavljaju se instrukcije sa predviđene lokacije. Ako se pokaže da je nagovještaj loš, bit se invertuje. Bafer se ponaša kao keš kod kojeg je svaki pristup pogodak, a performanse mu zavise od toga koliko često se predviđanje odnosi na aktuelno grananje, i kada se odnosi, koliko je precizno. Nezavisno od keširanja - otkrivanja aktuelnog grananja, jednobitna struktura predviđanja ima bitan nedostatak: iako se grananje gotovo uvijek dešava, biće pogrešna dva umjesto jednog grananja, kada se grananje ne desi.

Primjer: grananje u petlji koje se devet puta desi a jednom ne, će biti pogrešno predviđeno u prvom i zadnjem prolazu (u prvom jer je bit ostao u stanju negrananja od zadnjeg prolaza). Zato je tačnost predviđanja granja koje se dešava u 90% slučajeva jednaka 80%.

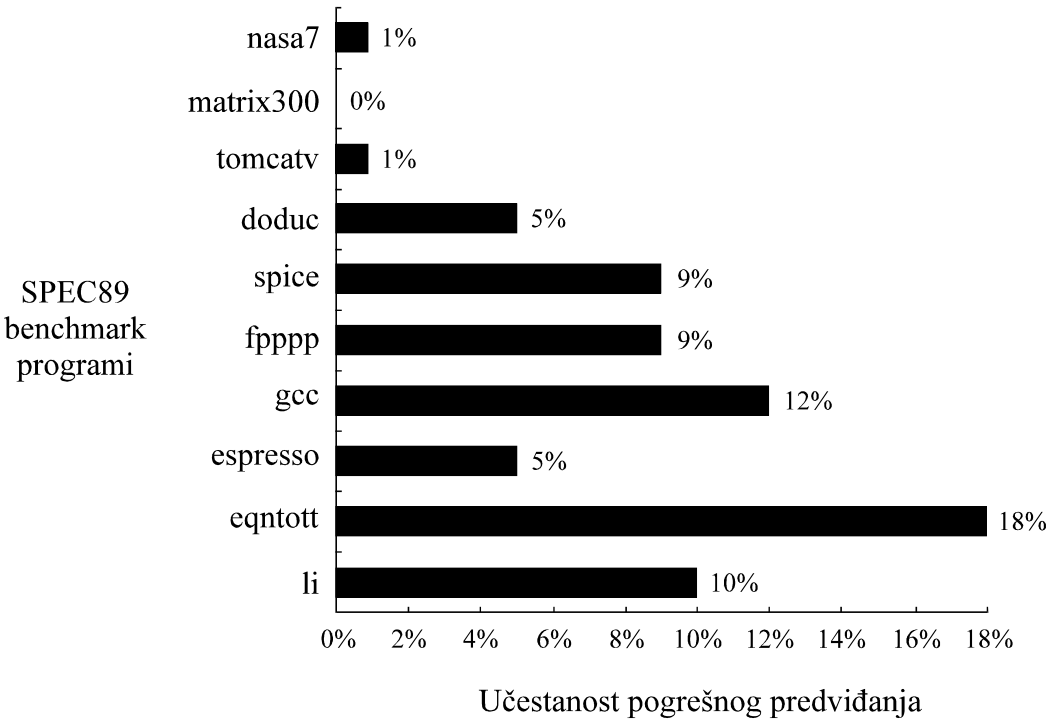
Zato se uvodi dvo-bitna struktura za predviđanje - tada se mora pogriješiti dva puta uzastopno, prije nego što se predviđanje promijeni. Slika 7.1. prikazuje dijagram prelaza strukture sa dvo-bitnim mehanizmom predviđanja.

Dvo-bitna struktura je specijalni slučaj n-bitne, koja ima n-bitni “brojač zasićenja” uz svako polje bafera za predviđanje grananja. Kada brojač ima vrijednost veću od polovine svoje maksimalne vrijednosti (2^{n-1}) predviđa se grananje, dok se u ostalim slučajevima predviđa “ne-grananje”. Kao i kod dvo-bitne strukture, grananje inkrementira a ne-grananje dekrementira brojač. Dvo-bitna struktura se, u praksi, pokazala vrlo efikasnom, pa je koristi značajan dio savremenih arhitektura.

Bafer za predviđanje grananja se može realizovati kao poseban “keš” kojem se pristupa adresom instrukcija u IF fazi PS-e, ili kao par bita pridodat svakom bloku keša instrukcija koji se dobavlja sa instrukcijama. Ako se instrukcija dekodira kao grananje i predviđa se da će se ono desiti, pribavljanje se nastavlja sa adrese na koju se grana, čim se odredi njena vrijednost (vrijednost PC-a). U protivnom nastavlja se sekvencijalno pribavljanje i izvršavanje. Ako se ustanovi da je predviđanje bilo netačno, biti predviđanja se mijenjaju kao na slici 7.1.



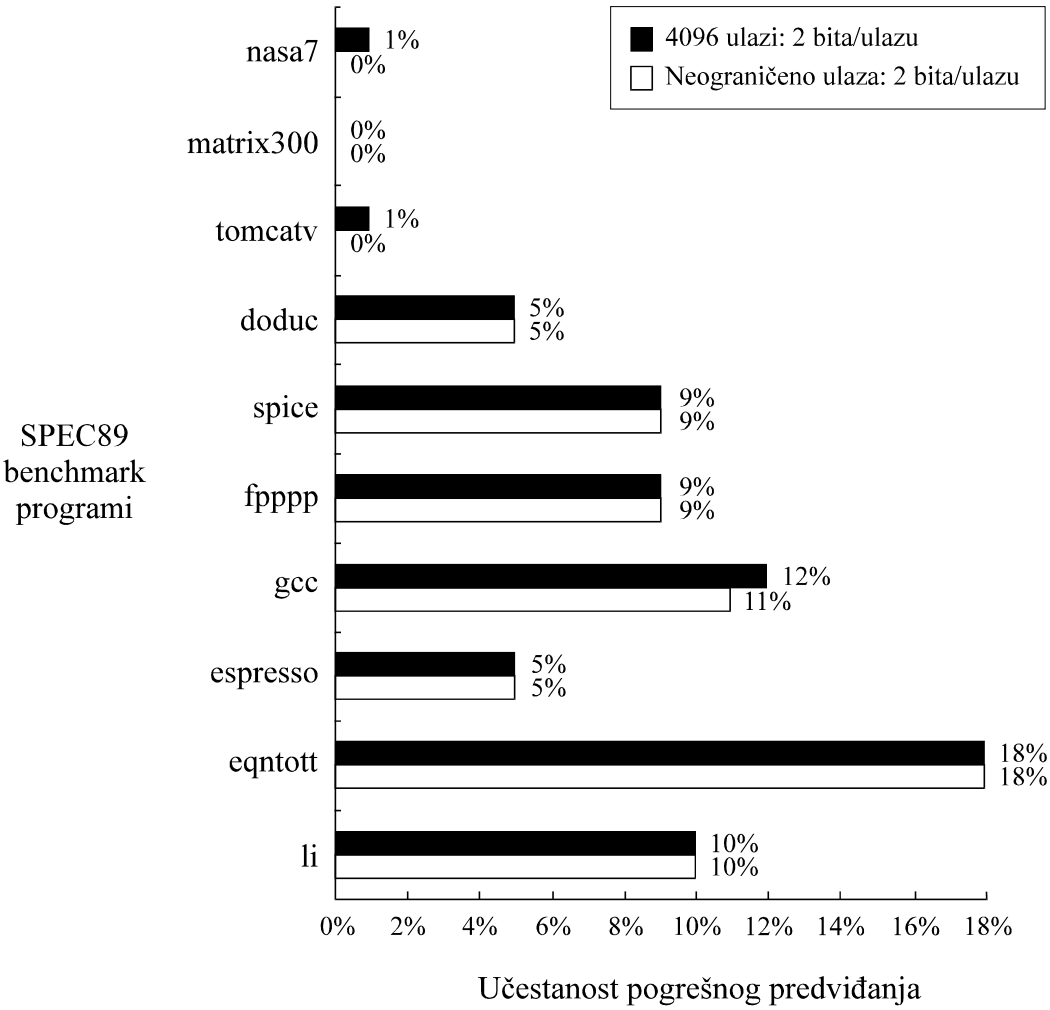
Slika 7.1. Stanja kod dvo-bitne šeme predviđanja grananja. Korištenjem dva umjesto jednog bita, grananje sa vjerovatnijim jednom od dva ishoda - a takva je većina - će biti pogrešno predviđena samo jednom. Ta dva bita kodiraju četiri stanja sistema.



Slika 7.2. Tačnost predviđanja sa 4096-ulaznim dvo-bitnim baferom za SPEC89 benčmark-programe. Učestanost pogrešnog predviđanja za cjelobrojne programe (gcc, espresso, eqntott i li) je značajno viša (prosjek 11%) od one kod FP programa (prosjek 4%).

Ogledna PS otkriva da li će biti grananja i adresu grananja skoro u isto vrijeme, pod uslovom da nema hazarda prilikom pristupa registru koji je specificiran u uslovu grananja, pa za jednostavnu arhitekturu ovaj mehanizam predviđanja nije od velike pomoći.

Neke ilustracije tačnosti predviđanja su date na slikama 7.2. i 7.3.



Slika 7.3. Tačnost predviđanja kod 4096-ulaznog dvo-bitnog bafera u odnosu na bafer sa neograničenim brojem ulaza za SPEC89 benchmark-programe

Tačnost predviđanja grananja je, u nekim slučajevima, moguće povećati vodeći računa o ponašanju drugih grananja u programu.

Primjer:

```
if (aa==2)
        aa=0;
if (bb==2)
        bb=0;
if (aa!=bb) {…
```

se može prevesti u kôd ogledne arhitekture, sa *aa* i *bb* u registrima R1 i R2, ovako:

```
        SUBI    R3, R1, #2      ; aa==R1; bb==R2
        BNEZ    R3, L1          ; grananje b1 (aa!=2)
        ADD     R1, R0, R0      ; aa==0
L1:     SUBI    R3, R2, #2
```

```
        BNEZ    R3, L2      ; grananje b2 (bb!=2)
        ADD     R2, R0, R0  ; bb==0
L2:     SUB     R3, R1, R2  ; R3=aa-bb
        BNEZ    R3, L3      ; grananje b3 (aa==bb)
```

Ponašanje grananja b3 je u vezi sa grananjima b1 i b2. Jasno je da ako se ne izvrše ni b1 ni b2, tada će se desiti b3. Korištenjem samo pojedinačnih mehanizama predviđanja ovo ponašanje se ne može "otkriti". Mehanizmi predviđanja koji vode računa o ponašanju drugih grananja se zovu vezani (eng. correlating) ili dvo-nivoski.

Evo još jednog slikovitog primjera:

```
if (d==0)
        d=1;
if (d==1)
```

se prevodi u asemblerski kod, uz pretpostavku da se d nalazi u R1, kao:

```
        BNEZ    R1, L1      ;grananje b1 (d!=0)
        ADDI    R1, R0, #1  ; d==0, pa je d=1
L1:     SUBI    R3, R1, #1
        BNEZ    R3, L2      ;grananje b2 (d!=1)
        …
L2:
```

Uz pretpostavku da *d* ima vrijednosti 0, 1 i 2, moguće sekvence izvršenja su date u tabeli 7.1. Ako se ne desi b1, neće se desiti ni b2. Mehanizam vezanog predviđanje ovo može iskoristiti.

Početna vrijednost d	d==0?	b1	Vrijednost d prije b2	d==1?	b2
0	Da	Bez grananja	1	Da	Bez grananja
1	Ne	Grananje	1	Da	Bez grananja
2	Ne	Grananje	2	Ne	Grananje

Tabela 7.1. Moguće sekvence izvršenja za dati segment koda.

Sekvenca u kojoj se *d* mijenja od 2 do 0, sa jedno-bitnim mehanizmom predviđanja inicijaliziranim na ne-grananje, se odvija kao u tabeli 7.2. - sva su grananja pogrešno predviđena!!!

d=?	b1 predviđanje	b1 akcija	Novo b1 predviđanje	b2 predviđanje	b2 akcija	Novo b2 predviđanje
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Tabela 7.2. Ponašanje jedno-bitnog predviđanja initializiranog na "Bez grananja". T znači grananje, NT znači bez grananja.

Drugo rješenje je mehanizam predviđanja grananja koji koristi jedan bit korelacije - vezanog predviđanja. Neka svako grananje ima dva bita za predviđanje - jedan za predviđanje pod

uslovom da se prethodno grananje nije desilo, a drugi ako se ono desilo. U opštem slučaju, zadnje grananje nije ista instrukcija grananja za koju se vrši predviđanje (osim kod jednostavnih petlji sa jednim grananjem).

Četiri moguće kombinacije i njihova značenja su data u tabeli 7.3..

Biti predviđanja	Predviđanje ako se zadnje grananje nije desilo	Predviđanje ako se zadnje grananje desilo
NT/NT	Bez grananja	Bez grananja
NT/T	Bez grananja	Grananje
T/NT	Bez grananja	Grananje
T/T	Grananje	Grananje

Tabela 7.3. Kombinacije i značenja bitâ predviđanja. T znači grananje, NT znači bez grananja.

Efekat jednobitnog mehanizma predviđanja grananja sa jednim bitom vezanog predviđanja, inicijaliziranog na ne-grananje/ne-grananje, je dat u tabeli 7.4.

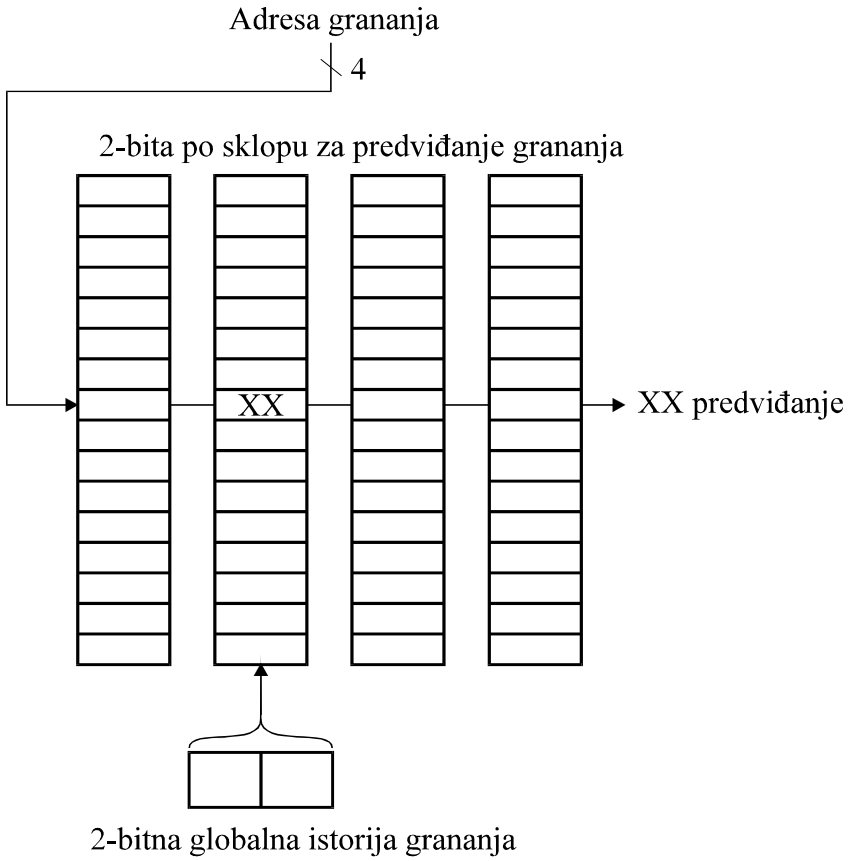
d=?	b1 predviđanje	b1 akcija	Novo b1 predviđanje	b2 predviđanje	b2 akcija	Novo b2 predviđanje
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Tabela 7.4. Efekat jednobitnog mehanizma predviđanja grananja sa jednim bitom korelacije, inicijaliziranog na ne-grananje/ne-grananje. T znači grananje, NT znači bez grananja. Uzeto predviđanje je prikazano podebljano.

U ovom slučaju, jedino pogrešno predviđanje je u prvoj iteraciji za d=2. Tačno predviđanje kod b1 se desilo zbog izbora vrijednosti d, jer b1 nije u jasnoj vezi sa prethodnim predviđanjem b2. S druge strane, korektno predviđanje b2, pokazuje prednosti vezanog predviđanja. Čak i za druge vrijednosti d, predviđanje za b2 bi korektno predvidilo slučaj kada se b1 nije granalo kod svakog izvršenja b2 nakon jednog početnog netačnog predviđanja.

Predviđanje u tabelama 7.3. i 7.4. se zove (1,1) mehanizam predviđanja jer koristi ponašanje zadnjeg grananja da izabere između para jedno-bitnih mehanizama predviđanja grananja. U opštem slučaju (m,n) mehanizmi koriste ponašanje zadnjih m grananja da izaberu između 2^m mehanizama predviđanja grananja, od kojih je svaki n-bitni mehanizam predviđanja za jedno grananje. Ovaj mehanizam vezanog predviđanja ne zahtijeva složenu hardversku podršku: ponašanje u zadnjih m grananja se može snimiti u m-bitni šift-registar - po jedan bit za svako grananje. Bafer za predviđanje se onda može indeksirati donjim bitima adrese instrukcije grananja povezanim sa m-bitnim šift-registrom. Slika 7.4. prikazuje (2,2) mehanizam predviđanja.

Kako bafer za predviđanje nije keš, brojači indeksirani jednom vrijednošću globalnog mehanizma predviđanja se mogu odnositi na različita grananja u određenom trenutku. Slika 7.4. prikazuje bafer kao dvodimenzionalni objekt. U realnosti, to može biti linearni niz memorijskih lokacija, širok dva bita. (2,2) bafer ima ukupno 64 podatka/ulaza - 4 najniža bita adrese grananja (adrese riječi) i dva bita koji prate istoriju grananja, čine 6-bitni indeks kojim se indeksira 64 brojača.



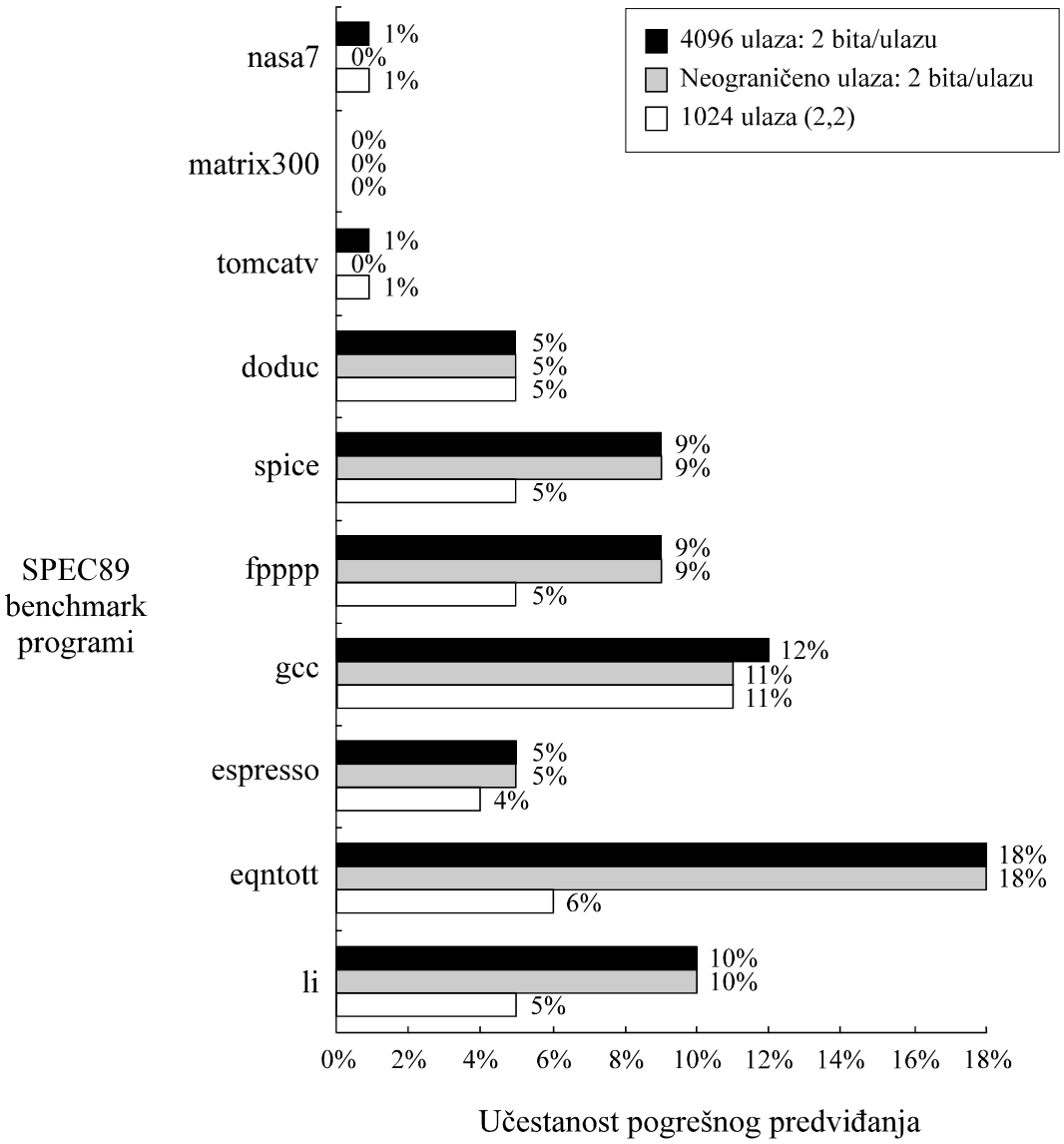
Slika 7.4. (2,2)-bafer za predviđanje granajna koristi dvo-bitnu globalnu istoriju da bi izabrao među četiri sklopa za predviđanje za svaku adresu grananja. Svaki sklop za predviđanje je - dvo-bitni prediktor samo za to grananje. Prikazani bafer za predviđanje grananja ima ukupno 64 ulaza; adresa grananja se koristi za izbor četiri od ovih ulaza a globalna istorija jedan od ta četiri. Dvo-bitna globalna istorija se može realizovati kao šift-registar koji jednostavno unosi ponašanje grananjačim se ono ustanovi.

Kako porediti mehanizme vezanog predviđanja sa standardnim dvo-bitnim mehanizmom? Kao kriterijum ravnopravnosti može biti uzet broj bita stanja koji se koriste. Njihov broj za (m,n)-mehanizam predviđanja je

$2^m \times n \times$ [broj parametara (ulaza) za predviđanje koje bira adresa instrukcije grananja]

Dvo-bitni mehanizam predviđanja bez pamćenja prošlih grananja je (0,2) mehanizam.

Za mehanizam na slici 7.4., to je $2^2 \times 2 \times 16 = 128$ bita. (0,2) mehanizam sa 4K parametara im $2^0 \times 2 \times 4K = 8K$ bita. (2,2) mehanizam sa istim brojem bita bi imao 1K parametara/ulaza. Poređenje ova dva mehanizma je dato na slici 7.5. (2,2) mehanizam je bolji i od jednostavnog dvo-bitnog mehanizma sa istim brojem bita stanja i od takvog mehanizma sa neograničenim brojem parametara.



Slika 7.5. Poređenje dvo-bitnih sklopova za poređenje. 4096 bitni prediktor bez korelacije je prvi, takav dvo-bitni sa neograničenim brojem ulaza je drugi i dvo-bitni prediktor sa dva bita globalne istorije sa ukupno 1024 ulaza je treći.

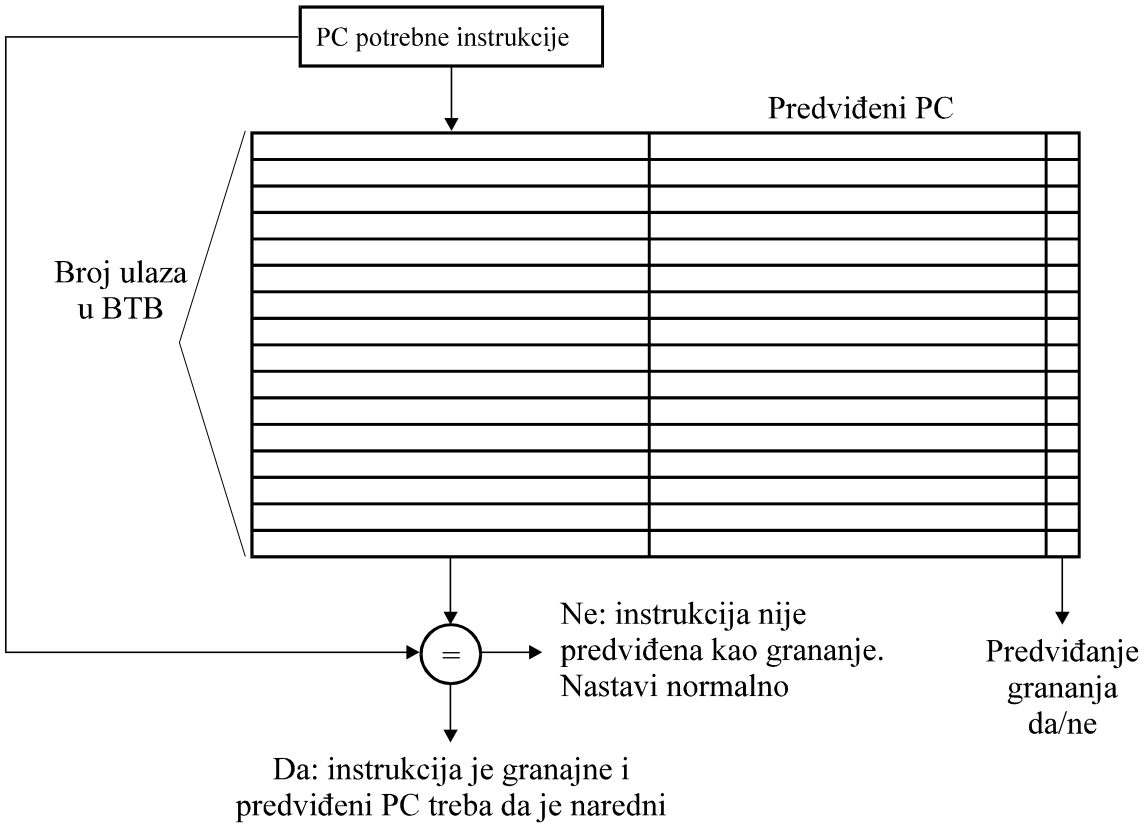
7.1. Korištenje bafera odredišnih adresa grananja

Da bi se smanjilo kašnjenje kod grananja kod ogledne arhitekture, treba znati odakle dobavljati sljedeću instrukciju na kraju IF faze - da li je dobavljena instrukcija grananje, i, ako jeste, koja je sljedeća vrijednost PC-a. Ako je to poznato, neće biti zastoja. Keš za predviđanje grananja koji sadrži i adresu sljedeće instrukcije nakon grananja se zove bafer odredišnih adresa grananja (BTB, od eng. Branch-Target-Buffer) ili keš odredišnih adresa (BTC, od eng. Branch-Target-Cache).

Kod standardne ogledne PS-e, baferu za predviđanje grananja se pristupa za vrijeme ID faze, pa je na njenom kraju poznata adresa grananja (izračunato u ID), adresa prolaza (izračunato u IF) i predviđanje grananja. Tako je, na kraju ID, poznato odakle treba dobaviti sljedeću (predviđenu) instrukciju. BTB-u se pristupa u IF fazi pomoću adrese upravo pribavljene instrukcije (možda grananja!). Ako se desi pogodak, tada je (predviđena) sljedeća adresa

poznata na kraju IF faze - jedan ciklus prije nego sa baferom za predviđanje grananja.

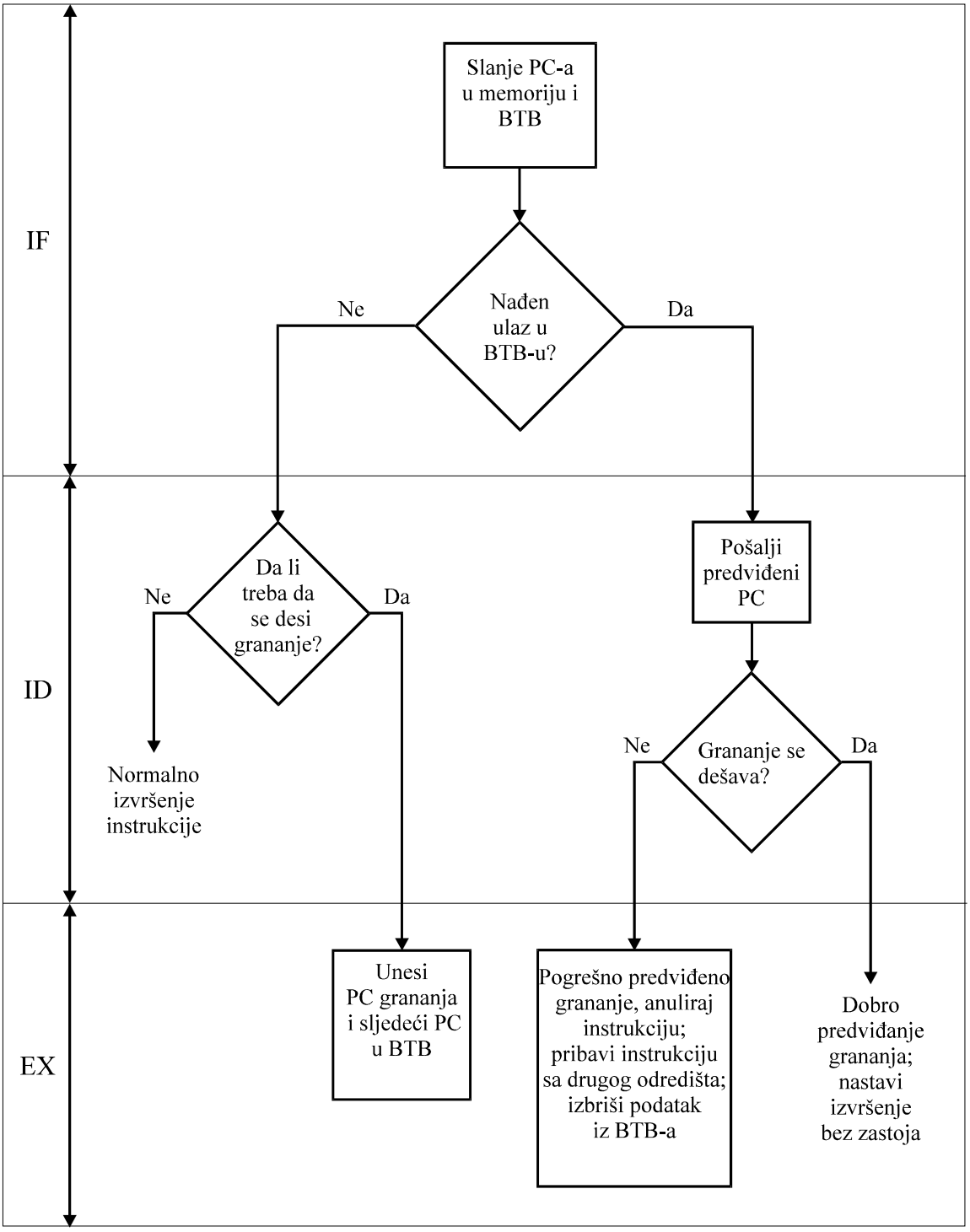
Slika 7.6. daje izgled BTB-a. Ako PC dobavljene instrukcije odgovara nekom iz bafera, tada se odgovoarajući predviđeni PC uzima kao adresa sljedeće instrukcije. Mehanizam je isti kao kod klasičnog keša.



Slika 7.6. Bafer odredišnih adresa grananja (BTB). PC instrukcije koja se dobavlja se poredi sa skupom adresa instrukcija u prvoj koloni; one predstavljaju adrese poznatih grananja. Ako PC odgovara jednoj od njih, tada se instrukcija koja se pribavlja smatra grananjem (koje će se desiti), a drugo polje, predviđeni PC, sadrži predviđanje slijedećeg PC-a nakon grananja. Treće polje je opcionalno i može se koristiti za dodatne bite stanja predviđanja.

U BTB-u je potrebno držati samo podatke o grananjima koja su se desila (i predviđa se njihovo buduće dešavanje), dok se ona koja se nisu desila (i ne predviđaju se) nisu potrebna - oni se tretiraju kao ostale instrukcije.

Slika 7.7. prikazuje faze/korake korištenja BTB-a i mjesta događanja u PS-i. Ako se desi pogodak u BTB-u i predviđanje je korektno - neće biti kašnjenja zbog grananja. U protivnom, javiće se kašnjenje od najmanje dva ciklusa (ili više, zbog ažuriranja BTB-a).



Slika 7.7. Koraci pri obradi instrukcije sa BTB-om. Ako je PC instrukcije nađen u baferu, tada ta instrukcija mora biti grananje za koje se predviđa da će se desiti; tada pribavljanje odmah počinje sa predviđenog PC-a u ID. Ako nije nađen a pokaže se kao grananje koje se desilo, unosi se u bafer zajedno sa ciljnom adresom, što je poznato na kraju ID. Ako je nađen, ali se pokaže da je instrukcija grananje koje se neće desiti, uklanja se iz bafera. Ako je instrukcija grananje, nađena i korektno predviđena, nastavlja se izvršenje bez zastoja. Ako predviđanje nije tačno, troši se dodatni ciklus (kašnjenje) za ponovno dobavljanje korektne instrukcije.

Tabela 7.5. daje podatke o kašnjenjima u svim mogućim slučajevima.

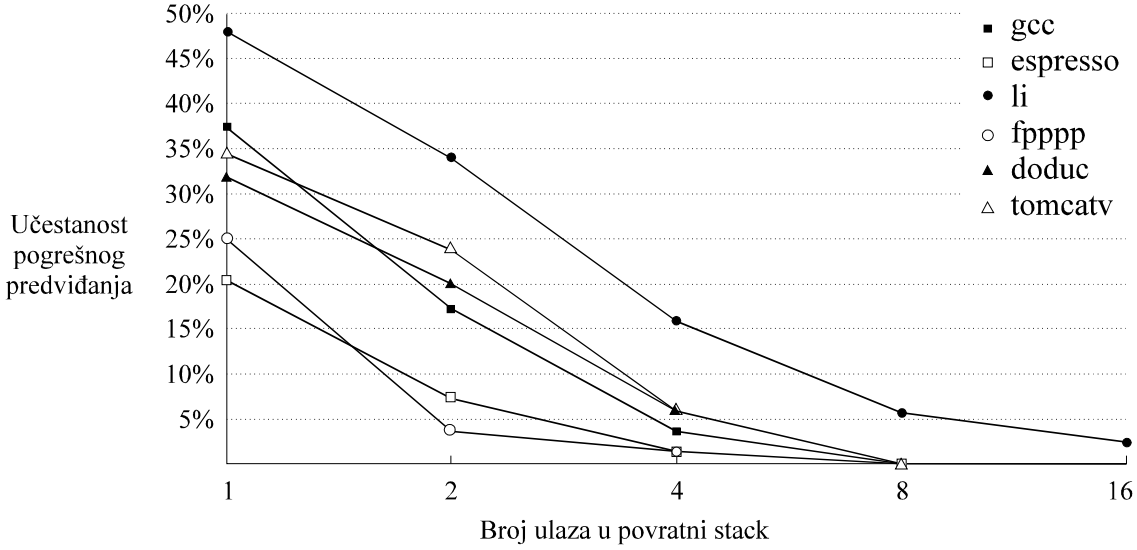
Instrukcija u baferu	Predviđanje	Stvarno granajne	Zastoj u ciklusima
Da	Grananje	Grananje	0
Da	Grananje	Bez grananja	2
Ne		Grananje	2

Tabela 7.5. Kašnjenja za sve moguće kombinacije - da li je grananje u baferu i šta se s njim dešava, uz pretpostavku da se u bafer upisuju samo grananja koja se dese. Nema kašnjenja ako je grananje nađeno u baferu i sve je korektno predviđeno. Ako grananje nije korektno predviđeno, kasni se jedan ciklus za upis korektnih informacija u bafer (za to vrijeme se ne može dobaviti instrukcija) i jedan ciklus, ako je potrebno, za ponovno donošenje sljedeće korektne instrukcije. Ako se pokaže da se grananje ne dešava, kašnjenje od dva ciklusa je neophodno za upis novih podataka u bafer.

Drugi način realizacije BTB-a bi bio da se umjesto, ili pored, adresa grananja, smještaju i njihove instrukcije (sa odredišne adrese). Time se omogućava:

1. da vrijeme pristupa BTB-u bude i duže od vremena između dva uzastopna pribavljanja instrukcija, pa bafer može biti veći, i
2. optimizacija - preklapanje grananja (branch folding) - 0-ciklusno bezuslovno i, ponekad, 0-ciklusno uslovno grananje.

Predviđanje indirektnih skokova (kada odredišna adresa varira u toku izvršenja) pretstavlja poseban problem. Takvi skokovi se, najčešće, dešavaju kod indirektnih poziva procedura i kod povrataka iz procedura. Tačnost predviđanja povrataka iz procedura pomoću BTB-a može biti mala ako se procedura poziva sa više mjesta i nikad sa jednog mjesta više puta uzastopno. Tada je korisno baferovati povratne adrese u strukturi sličnoj steku. Tada, ako je keš dovoljno velik, potpuno tačno će se predviđati povratne adrese. Slika 7.8. ilustruje performanse takve strukture sa 1 do 16 elemanata u povratnom baferu (steku).



SLIKA 7.8. Tačnost predviđanja kada bafer povratne adrese radi kao stack. Tačnost je dio korektno predviđenih povratnih adresa. Prosječna tačnost je 81% indirektnih skokova u datih šest benchmark - programa.

Tehnike predviđanja grananja su ograničene svojom tačnošću i kašnjenjima kod pogrešnog predviđanja. Tipična tačnost se kreće od 80-95%, zavisno od tipa programa i veličine bafera. Uz povećanje tačnosti, neophodno je smanjiti kašnjenja kod pogrešnog predviđanja. To se postiže dobavljanjem instrukcija sa obje lokacije - grananja i ne-grananja. Za to je neophodna memorija sa dva pristupa, keš sa preplitanjem, ili dobavljanje sa jednog, pa sa drugog mjesta.

Sve ovo povećava troškove izgradnje sistema, ali može biti jedini način da se smanje kašnjenja ispod određene granice.

8. Podrška kompajlera u povećanju paralelizma na nivou instrukcija

8.1. Otkrivanje i otklanjanje zavisnosti

Pronalaženje zavisnosti u programu je važno pri:

1. raspoređivanju instrukcija u kodu,
2. određivanju paralelizma u petljama i
3. rješavanju zavisnosti imena.

Složenost analiza zavisnosti proizilazi iz prisustva nizova i pokazivača u jezicima kao što je C. Pošto se referenciranje skalarnih vrijednosti eksplicitno odnosi na ime (registar ili memorijska adresa), oni se mogu analizirati relativno jednostavno.

Primjer:

```
for (i=1; i<=100; i=i+1) {
        A[i]=B[i]+C[i];
        D[i]=A[i]*E[i];
}
```

Pošto zavisnost A niza ne zavisi od petlje, moguće je petlju "odmotati" i pronaći paralelizam - samo se ne smije zamijeniti dva pristupa A-nizu. Količina paralelizma je ograničena brojem "odmotavanja", koje je ograničeno brojem iteracija petlje. Da bi se iskoristio toliki paralelizam, bilo bi potrebno mnogo FJ-a i registara.

U prethodnom kodu, drugi pristup A-nizu se ne mora prevesti u load-instrukciju, jer je prethodna izračunata vrijednost upisana u memoriju, pa se drugi pristup može ograničiti na registar u kome je vrijednost izračunata. Analiza zavisnosti podataka, obično, kaže da zavisnost može postojati, dok složenije analize treba da potvrde da se ovakva optimizacija može obaviti (da se u oba pristupa A-u pristupa istoj lokaciji u memoriji).

Zavisnosti u petlji su često u formi rekurencije (eng. recurrence) - povratka, kao u sljedećem primjeru:

```
for (i=2; i<=100; i=i+1) {
        Y[i]=Y[i-1]+Y[1];
}
```

Rekurencijom se naziva definisanje varijable na osnovu njene vrijednosti u prethodnim iteracijama petlje. Njeno otkrivanje može biti važno iz dva razloga - prvi, zbog toga što neke arhitekture (posebno vektorski računari) imaju posebnu podršku rekurenciji, a drugi, što neke rekurencije mogu biti značajan izvor paralelizma, kao u sljedećem primjeru:

```
for (i=6; i<=100; i=i+1) {
        Y[i]=Y[i-5]+Y[i];
}
```

U i-toj iteraciji referencira se i-5-ti elemenat, pa se kaže da petlja ima distancu zavisnosti 5. Što je ova distanca veća, to se više paralelizma može postići "odmotavanjem" petlje.

Kako kompajler otkriva zavisnosti? Skoro svi algoritmi za analizu zavisnosti rade sa pretpostavkom da se nizovima pristupa indeksima koji imaju formu:

$$a \times i + b$$

gdje su a i b konstante a i - indeks petlje. Određivanje zavisnosti između dva pristupa istom nizu u petlji, tada odgovara određivanju da li postoje iste vrijednosti dvaju takvih izraza za svako i u granicama petlje.
Pored otkrivanja zavisnosti, kompajler nastoji odrediti njen tip, kako bi je, u slučaju zavisnosti imena, mogao eliminisati reimenovanjem i kopiranjem.

Primjer - petlja koja ima više vrsta zavisnosti:

```
for (i=1; i<=100; i=i+1) {
        Y[i] = X[i] / c;        /* S1 */
        X[i] = X[i] + c;        /* S2 */
        Z[i] = Y[i] + c;        /* S3 */
        Y[i] = c - Y[i];        /* S4 */
}
```

S1 i S3 kao i S1 i S4 su zavisne zbog Y[i]. To nisu zavisnosti petlje, pa se ona može smatrati paralelnom. S3 i S4 će čekati na završetak S1.

Postoji antizavisnost S1 i S2 zbog X[i].

Postoji antizavisnost S3 i S4 zbog Y[i].

Postoji izlazna zavisnost S1 i S4 zbog Y[i].

Petlja koja eliminiše ove zavisnosti je:

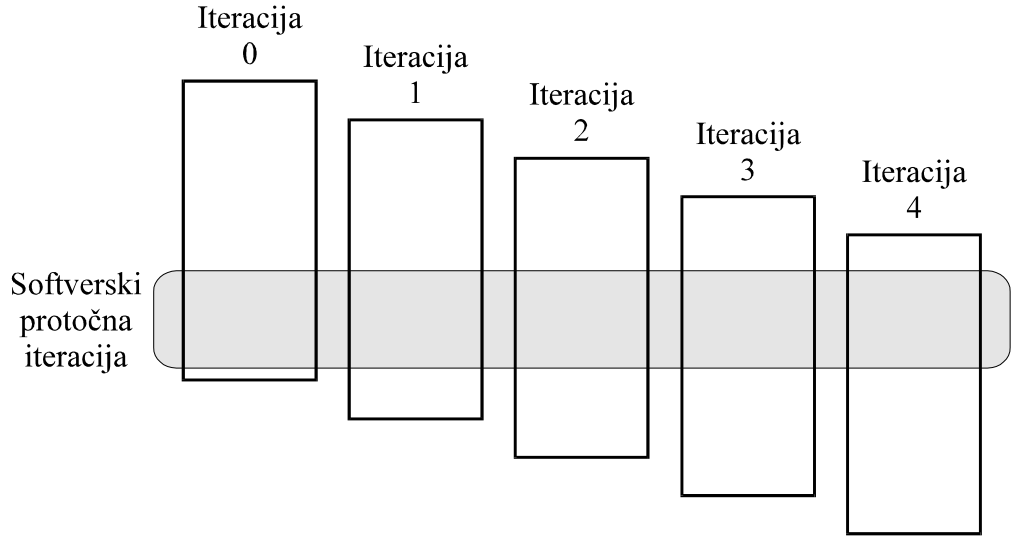
```
for (i=1; i<=100; i=i+1) {
        T[i] = X[i] / c;        /* Y reimenovano u T zbog izlazne zavisnosti */
        X1[i] = X[i] + c;       /* X reimenovano u X1 zbog antizavisnosti */
        Z[i] = T[i] + c;        /* Y reimenovano u T zbog antizavisnosti */
        Y[i] = c - Y[i];
}
```

Poslije izvršenja petlje X je reimenovan u X1, pa u kodu koji slijedi kompajler može zamijeniti ime X sa X1 i izbjeći kopiranje. U suprotnom, neophodno je kopiranje.
Analiza zavisnosti je osnovni način iskorištavanja paralelizma, i na nivou instrukcija (raspoređivanje pristupa memoriji) i na nivou petlji (“odmotavanje”). Njen glavni nedostatak je što se može primjeniti samo u određenim situacijama - kod jednostruko ugnježdenih petlji i kod pristupa nizovima na ranije spomenuti način. Neke od situacija u kojima analiza zavisnosti ne pomaže su:

1. kada se pristupa objektima pokazivačima (umjesto indeksima),
2. kada je indeksiranje niza indirektno (npr. kroz drugi niz),
3. kada zavisnost postoji za neke vrijednosti ulaza, ali se one u realnosti ne javljaju i
4. kada optimizacija zavisi ne samo od mogućnosti postojanja zavisnosti, već npr. od kojeg pisanja varijable zavisi njeno čitanje.

8.2. Softverske protočne strukture

Softverska protočnost je način reorganizacije petlji (simboličko odmotavanje) takve da se svaka iteracija sastoji od instrukcija izabranih iz različitih iteracija originalne petlje. U primjeru na slici 8.1 raspoređivač prepliće instrukcije iz različitih iteracija petlje, da bi razdvojio zavisne instrukcije iz jedne iteracije. Softverski protočna petlja prepliće instrukcije iz različitih iteracija bez odmotavanja petlje, kao na primjeru koji slijedi. Tu se softverski odrađuje ono što kod Tomasulovog algoritma radi hardver. Petlja u tom slučaju sadrži jedno čitanje iz memorije, jedno sabiranje i jedno smještanje u memoriju, svako iz različite iteracije. Postoji neophodni dodatni početni i završni kod za prilagođenje ovakvom izvršavanju.



SLIKA 8.1 Petlja softverske protočnosti bira instrukcije iz različitih iteracija petlje, tako razdvajajući zavisne instrukcije unutar jedne iteracije originalne petlje. "Uvodni" i "zaključni" dio koda odgovara dijelovima iznad i ispod softverski protočne iteracije.

Primjer: petlja koja inkrementira elemente niza čija je početna adresa u R1 konstantom u F2 je:

```
Loop: LD     F0, 0(R1)
      ADDD   F4, F0, F2
      SD     0(R1), F4
      SUBI   R1, R1, #8
      BNEZ   R1, Loop
```

Softverska protočnost simbolično odmota petlju i izabere instrukcije iz svake od njih. Kako je

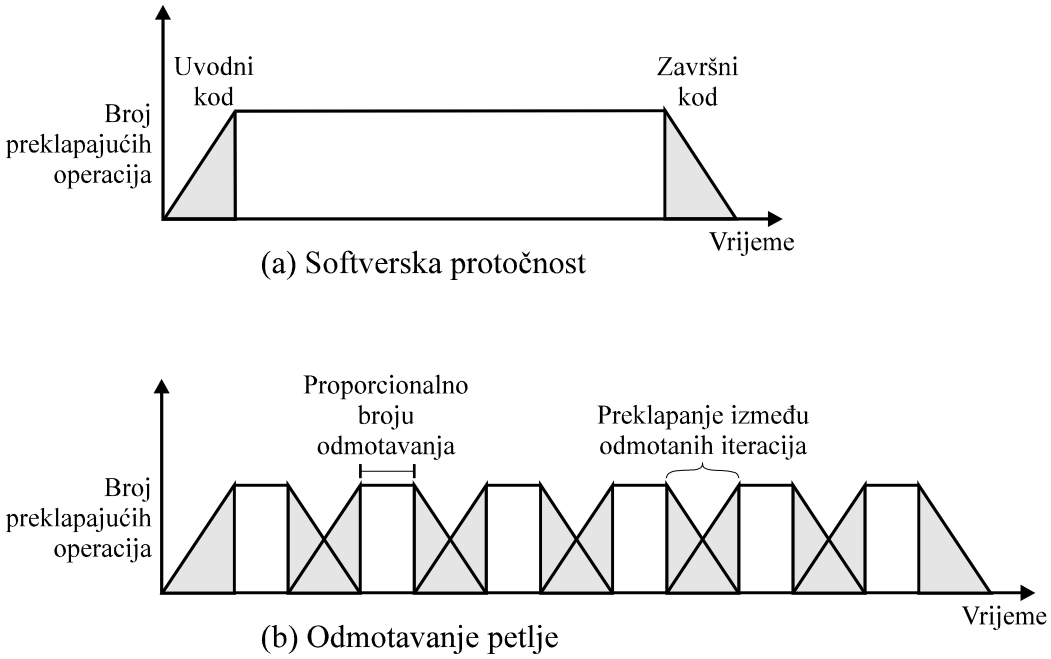
odmotavanje simbolično, instrukcije za upravljanje petljom (SUBI i BNEZ) se ne moraju replicirati. Odmotane petlje i izabrane instrukcije izgledaju ovako:

```
i-ta iteracija   LD                F0, 0(R1)
                 ADDD   F4, F0, F2
                 SD     0(R1), F4   ;izabrana
i+1-a iteracija  LD                F0, 0(R1)
                 ADDD   F4, F0, F2   ;izabrana
                 SD     0(R1), F4
i+2-a iteracija  LD                F0, 0(R1)   ;izabrana
                 ADDD   F4, F0, F2
                 SD     0(R1), F4
```

Izabrane instrukcije zajedno sa instrukcijama za upravljanje petljom čine novu petlju:

```
Loop:  SD     0(R1), F4     ; smješta u M[i]
       ADDD   F4, F0, F2    ; dodaje sadržaju M[i-1]
       LD     F0, -16(R1)   ; čita M[i-2]
       SUBI   R1, R1, #8
       BNEZ   R1, Loop
```

Ignorišući početni i završni kod, petlja se može izvršavati brzinom od 5 ciklusa po rezultatu. Kako se čita M[i-2], petlja ima dvije iteracije manje - ostalo treba da odradi pripremni i završni dio koda. Zbog uzastopnog korištenja istih registara (npr. F4, F0 i R1), od hardvera se očekuje da razriješi WAR hazarde u petlji, što u ovom slučaju nije problem jer nema zastoja zbog zavisnosti podataka.
Glavna prednost ovakvog pristupa u odnosu na klasično odmotavanje petlji je u zauzimanju manje prostora u memoriji. Obje tehnike, pored poboljšanja rasporeda instrukcija unutar petlje, svaka smanjuje različite neophodne dodatne poslove. Odmotavanje petlji smanjuje broj instrukcija za održavanje petlji, dok softverska protočnost smanjuje vrijeme kada se petlja ne izvršava maksimalnom brzinom na početku i na kraju petlje (slika 8.2.). Zato se najbolji rezultati postižu kombinovanjem ove dvije metode.

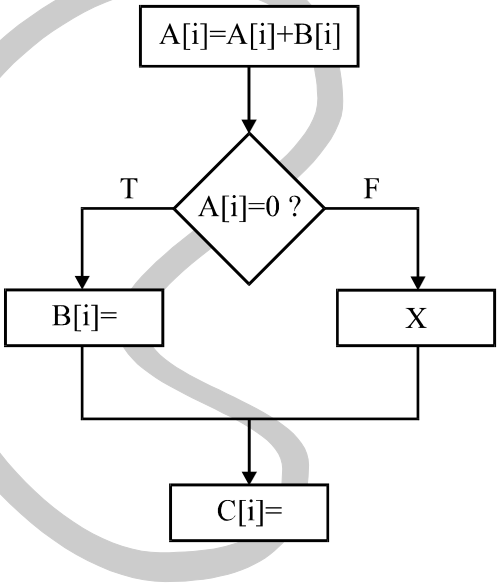


Slika 8.2. Izvršavanje petlje pomoću (a) softverske protočnosti i (b) odmotavanjem petlje. Osjenčena područja su vrijeme kada se petlja ne izvršava maksimalnom preklapanjem ili paralelizmom među instrukcijama. To se događa jednom na početku i jednom na kraju softverski protočne petlje. Kod odmotane petljeto se javlja m/n puta ako petlja ima ukupno m iteracija i odmotana je n puta. Svaki blok predstavlja jednu od n odmotanih iteracija. Povećanjem broja odmotavanja se smanjuje količinu uvodnih i završnih sekvenci. One se međusobno preklapaju i tako smanjuju vrijeme izvršenja.

8.3. Trasiranje

Još jedan način povećanja paralelizma je trasiranje (eng. trace scheduling). On proširuje odmotavanje petlji pronalazeći paralelizam i van grananja (ne samo grananja u petlji). Trasiranje je korisno kod procesora sa velikim brojem pokretanja instrukcija po ciklusu, gdje samo odmotavanje petlji ne daje dovoljno paralelizma za zapošljavanje svih FJ u procesoru. Trasiranje se sastoji od dva odvojena procesa. Prvi proces, izabiranja trase, pokušava naći vjerovatnu sekvencu osnovnih blokova koji će se objediniti u manjii broj instrukcija - trasu. Odmotavanje petlji sa velikom vjerovatnoćom predviđanja grananja se koristi za generisanje dugih trasa - sekvenci povezanih osnovnih blokova (bez grananja). Drugi proces je sabijanje trase (trace compaction) u što manji broj širokih instrukcija - paketa za pokretanje. Tom prilikom se instrukcije premiještaju tako da se pokrenu što je prije moguće.
Sabijanje trase je globalno raspoređivanje kôda, gdje se kôd sabija u najkraću moguću sekvencu uz očuvanje upravljačkih i zavisnosti podataka. Zavisnosti podataka diktiraju lokalno međusobni raspored instrukcija, dok upravljačke zavisnosti određuju instrukcije preko kojih se ostale instrukcije (kôd) ne mogu lako prebacivati. Zavisnosti podataka se rješavaju odmotavanjem i analizom referenciranja podataka da se ustanovi da li se dva pristupa podacima odnose na istu adresu. Upravljačke zavisnosti se isto tako smanjuju odmotavanjem. Glavna prednost trasiranja u odnosu na raspoređivanje u PS-i je u tome što smanjuje uticaj upravljačkih zavisnosti premiještanjem kôda preko uslovnih grananja (ne u petlji) korištenjem predviđenih ponašanja tih grananja. Iako ovakva premiještanja ne mogu garantovati ubrzanje, ako je tačno predviđanje grananja, kompajler može odrediti da li takvo

premiještanje ima šanse da dovede do bržeg izvršenja kôda. Slika 8.3. pokazuje fragment kôda, koji se može smatrati iteracijom odmotane petlje i izabrana trasa.



Slika 8.3. Fragment koda i odabrana trasa osjenčena sivom. Ova trasa bi bila prva izabrana ako je vjerovatnoća T grananja mnogo veća od F grananja. Grananje iz odluke (A[i]=0) ka X je grananje izvan trase, a grana od X do dodjele vrijednosti C-u je grananje u trasu. Ova grananja čine sažimanje-kompaktiranje trase teškim.

Kada se odabere trasa, ona se mora spakovati - komprimirati, tako da popunjava resurse procesora. To uključuje i premještanje instrukcija za dodjelu varijabli B i C u dio prije grananja. Premiještanje koda za dodjelu varijable B je spekulativno - ubrzaće izvršenje samo ako izvršenje krene tim putem. Svako globalno raspoređivanje instrukcija, uključujući i trasiranje, vrši ovakva premiještanja pod određenim ograničenjima. Kod trasiranja se grananja smatraju skokovima u i iz izabrane trase - najvjerovatnijeg puta. Kada se kôd premiješta preko ovih ulazno/izlaznih tačaka, dodatni “knjigovodstveni” kôd može biti neophodan na ulazu ili izlazu. Pod uslovom da je odabrana trasa najvjerovatniji put izvršenja, dodatni kôd se pokazuje efikasnim. Premiještanje koda mijenja i upravljačke zavisnosti, pa dodatni kod služi za zadržavanje dinamičke zavisnosti podataka. U slučaju premiještanja kôda za varijablu C, cijena “knjigovodstva” je jedina dodatna cijena, jer se taj kôd izvršava nezavisno od grananja. Kod spekulativnog premiještanja kôda, treba voditi računa da se ne omoguće pojave novih izuzetaka. To kompajler postiže tako što ne prebacuje neke vrste instrukcija, kao što su pristupi memoriji, koje mogu izazvati izuzetke.
Pod uslovom da se adrese varijabli A, B i C drže u R1, R2 i R3, sljedeća sekvenca instrukcija odgovara dijagramu toka sa slike 6.3:

```
LW      R4, 0(R1)          ; čitanje A
LW      R5, 0(R2)          ; čitanje B
ADDI    R4, R4, R5         ; sabiranje sa A
SW      0(R1), R4          ; pisanje A
```

```
            . . .
            BNEZ    R4, else_dio                    ; test A
            . . .                   ; then_dio
            SW      0(R2), …                        ; pisanje B
            j       nastavak                        ; skok preko else dijela
else_dio:   . . .                                     ; else dio
            X                                       ; kod za X
            . . .
nastavak:   . . .                                     ; poslije if-a
            SW      0(R3), …                        ; pisanje C
```

Kako je B upravljački zavisno od grananja prije premiještanja, ali ne i poslije, potrebno je obezbijediti da izvršenje iskaza ne može izazvati izuzetak, jer se on nebi desio u originalnom programu ako je else dio iskaza izabran. Premiještanje B ne smije uticati na tok podataka, jer bi to uticalo na krajnje izračunate vrijednosti.
Premiještanje B bi izazvalo promjenu toka podataka u programu, ako se pristupa B-u prije upisa (dodjeljivanja vrijednosti) bilo u X ili poslije *if* iskaza. U oba slučaja, premiještanje upisa u B će izazvati da neke instrukcije "i" (bilo u X ili kasnije u programu) postanu zavisne od podataka od premještene verzije upisa u B, umjesto ranijeg upisa u B koje se javlja prije petlje i od kojeg je "i" ranije (originalno) zavisilo.
Premiještanje upisa u C prije grananja zahtijeva dva koraka. U prvom, iskaz se premješta preko mjesta na kome "else" dio ulazi u trasu, u dio koji odgovara "then" dijelu. To čini instrukcije za C upravljački zavisnim od grananja i znači da se neće izvršiti u "else" dijelu (koji nije dio trase) ako se izabere. Prema tome, to utiče na instrukcije koje su bile zavisne od podataka od upisa u C i izvršavaju se poslije ovog dijela koda. Da bi se sačuvala korektnost izračunatih vrijednosti u takvim instrukcijama, pravi se kopija instrukcija koje računaju i upisuju C, u granu koja ulazi u trasu, na kraj X u "else" grani. U drugom, C se može premjestiti iz "then" grane preko uslovnog grananja, ako to ne utiče na tok podataka u uslovno grananje. Ako se C prebaci prije "if" iskaza njegova kopija u "else" grani postaje redundantna.

Odmotavanje petlji, softverska protočnost i trasiranje su tri tehnike koje imaju za cilj povećanje paralelizma na nivou instrukcija, koji bi se iskoristio kod procesora koji pokreću više od jedne instrukcije po ciklusu sata.

8.4. Podrška hardvera u povećanju paralelizma na nivou instrukcija

Odmotavanje petlji, softverska protočnost i trasiranje se mogu koristiti za povećanje paralelizma na nivou instrukcija kada je ponašanje grananja značajno predvidivo u trenutku kompajliranja. U suprotnom su neophodni drugi načini.
Prvi je proširenje skupa instrukcija dodavanjem uslovnih ili zasnovanih (eng. predicated) instrukcija, koje se mogu koristiti za eliminisanje grananja i pomoć kompajleru da prebacuje instrukcije preko grananja.
Drugi je spekulativno izvršavanje instrukcija - izvršavanje prije nego što procesor zna da li ih treba ili ne izvršavati i, na taj način, izbjegavanje zastoja usljed upravljačih zavisnosti.

8.4.1. Uslovne instrukcije

Uslovne instrukcije se izvršavaju samo ako je ispunjen dati uslov. U protivnom se izvršenje završava nakon otkrivanja da uslov nije ispunjen - kao da se izvršava NOP (od en. No operation) instrukcija. Najčešći primjer takvih instrukcija je uslovno premještanje sadržaja iz jednog registra u drugi. Takva instrukcija može eliminisati potrebu za grananjem u jednostavnijim sekvencama kôda i time unaprijediti rad PS-e.
Primjer:

```
if (A==0) {S=T;}
```

ako su varijable A, S i T u registrima R1, R2 i R3 - iskaz se može prevesti u asemblerski kôd sa grananjem:

```
BNEZ  R1, L
MOV   R2, R3
L:
```

dok se korištenjem uslovnog premještanja, koje se izvrši samo ako je treći operand jednak nuli, ovo može obaviti u jednoj instrukciji:

```
CMOV        R2, R3, R1
```

Ovim je upravljačka zavisnost iz kôda sa grananjem pretvorena u zavisnost podataka. Mjesto razrješavanja zavisnosti se ovim pomjera sa (blizu) početka (sa grananjem) na kraj PS-e, kada se obavlja upis u registre.
Uslovne instrukcije instrukcije se mogu koristiti za poboljšanje raspoređivanja instrukcija kod superskalarnih ili procesora sa vrlo dugom instrukcijskom riječju VLIW (eng.**Very long instruction word)** procesora pomoću spekulativnog izvršavanja instrukcija - npr. spekulativno premještanje vremenski kritičnih instrukcija.
Primjer - data je sekvenca kôda (tabela 8.1.) superskalarnog procesora koji može pokrenuti dvije instrukcije po ciklusu - jednog pristupa memoriji i jedne ALU-operacije, ili jedno grananje:

Prva instrukcija	Druga instrukcija
LW R1, 40(R2)	ADD R3, R4, R5
	ADD R6, R3, R7
BEQZ R10, L	
LW R8, 20(R10)	
LW R9, (R8)	

Tabela 8.1. Sekvenca paralelnog kôda superskalarnog procesora

U ovoj skvenci je izgubljeno vrijeme za jedan pristup memoriji, u drugom ciklusu. Zastoj zbog zavisnosti podataka će nastati ako se ne desi grananje jer zadnja LW operacija zavisi od prethodne.
Uvođenjem uslovnog čitanja riječi iz memorije (LWC), koje se izvršava samo ako je treći operand jednak nuli, može se realizovati sekvenca kao u tabeli 8.2.

Prva instrukcija	Druga instrukcija
LW R1, 40(R2)	ADD R3, R4, R5
LWC R8, 20(R10), R10	ADD R6, R3, R7
BEQZ R10, L	
LW R9, (R8)	

Tabela 8.2. Skraćena sekvenca kôda superskalarnog procesora

Ovakva sekvenca ubrzava izvršenje jer eliminiše jedno pokretanje instrukcija i reducira zastoj PS-e zbog zadnje instrukcije. Ova transformacija je spekulativna jer, ako kompajler pogrešno predvidi smjer grananja, uslovna instrukcija neće uticati na brzinu izvršavanja sekvence.
Prilikom ovakve upotrebe uslovnih instrukcija neophodno je obezbijediti da spekulativno izvršena instrukcija ne izazove neki izuzetak - da instrukcija nema nikakvog efekta ako uslov nije ispunjen.
Korist od uslovnih instrukcija je ograničena slijedećim faktorima:

1. uslovne instrukcije čiji uslovi nisu ispunjeni, ipak troše procesorsko vrijeme,
2. uslovne instrukcije su najkorisnije kada se uslov može ispitati ranije u PS-i; ako se uslov i grananje ne mogu razdvojiti (zbog zavisnosti podataka kod određivanja uslova), biće manja korist od uslovnih instrukcija, iako one odgađaju trenutak u kome se mora znati rezultat ispitivanja uslova, do blizu kraja PS-e,
3. upotreba uslovnih instrukcija je ograničena kod višestrukih uzastopnih grananja, jer je tada potrebno znati rezultat više uslova i logički ih pomnožiti prije izvršenja uslovne instrukcije,
4. uslovne instrukcije mogu usporavati rad procesora u odnosu na bezuslovne - više ciklusa po instrukciji ili nižu frekvenciju signala sata - pa ih, u tom slučaju, treba pažljivo koristiti.

Većina savremenih arhitektura koristi nekoliko jednostavnih uslovnih instrukcija - najčešće uslovno prebacivanje podatka iz registra u registar.

8.5. Kompajlersko spekulisanje uz podršku hardvera

U slučajevima kada se grananja u programu mogu predvidjeti u trenutku kompajliranja, kompajler može spekulisati da poboljša raspoređivanje ili poveća učestanost pokretanja instrukcija. Uslovne instrukcje omogućavaju ograničeno spekulisanje, ali su mnogo korisnije kada se upravljačke zavisnosti mogu potpuno eliminisati, kao kod “if-then” iskaza sa kratkom “then” granom. U pokušaju spekulacije kompajler nastoji ne samo da učini instrukcije upravljački nezavisnim, već i da ih premjesti unaprijed, tako da se spekulisane instrukcije izvrše prije grananja.
Pri prebacivanju instrukcija preko grananja, kompajler mora obezbijediti da se ne promijene

mogućnosti nastanka izuzetaka i da dinamička zavisnost podataka ostane nepromijenjena (kao što je prikazano kod trasiranja).
Ako kompajler ne prebacuje instrukcije koje mogu da izazovu izuzetke (pristupi memoriji i većina FP-operacija), time značajno smanjuje moguća ubrzanja. U svakom slučaju, potrebno je postići da se rezultati spekulativno izvršene sekvence koja je pogrešno predviđena, ne koriste u računanju konačnih rezultata.
Postoje tri metode za podršku spekulativnom izvršavanju, bez mijenjanja mogućnosti nastanka izuzetaka:

1. Hardver i OS zajedno ignorišu izuzetke nastale tokom spekulativnog izvršenja instrukcija.
2. Poseban skup statusnih bita "otrova" (eng. poison bits), se dodaje registrima u koje se upisuju rezultati spekulativnih instrukcija kada one izazovu izuzetak. Ti biti izazivaju-signaliziraju grešku kada "normalna" instrukcija pokuša koristiti taj registar.
3. Dodaje se mehanizam za označavanje spekulativnih instrukcija, pa se rezultati tih instrukcija baferuju sve do trenutka kada se sa sigurnošću može reći da su one prestale biti spekulativne.

Potrebno je razlikovati "neprolazne" izuzetke koji ukazuju na grešku u programu i, normalno, izazivaju kraj njegovog izvršenja (npr. "memory protection violation") i one "prolazne" koji, nakon obrade, dozvoljavaju nastavak izvršenja programa (npr. "page fault"). Instrukcije koje mogu izazvati prolazni izuzetak se mogu koristiti za spekulativno izvršavanje kao normalne instrukcije. Ako je spekulativno izvršenje pogrešno predviđeno, obrada nepotrebnih izuzetaka samo negativno utiče na performanse i ne može izazvati greške u izvršenju. Neprolazni izuzeci se ne bi smjeli desiti u korektno napisanim programima, osim kada se oni izvršavaju u svrhu debagiranja (eng. debugging mode). Ako se takav izuzetak javi kod spekulativne instrukcije, ne može se prihvatiti-obraditi izuzetak sve dok instrukcija ne prestane biti spekulativna.

8.6. Saradnja hardvera i softvera u svrhu spekulacije

U najjednostavnijem slučaju, hardver i operativni sistem jednostavno obrađuju sve prolazne izuzetke, kada se oni dese, i jednostavno vraćaju neodređene vrijednosti za svaki neprolazan izuzetak. Ako instrukcija koja je izazvala neprolazan izuzetak nije spekulativna, program izaziva grešku. Umjesto prekida programa, dozvoljava se njegov nastavak, iako je jasno da će, najvjerovatnije, dati pogrešan rezultat. Ako je instrukcija koja je izazvala neprolazan izuzetak spekulativna, program može biti korektan i spekulativni rezultat neće biti korišten, pa vraćanje neodređene vrijednosti ne može biti opasno. Ova metoda ne može izazvati grešku u korektnom programu, bez obzira na nivo spekulisanja. Nekorektan program, koji je ranije izazivao neprolazan (jedan ili više) izuzetak, će dati netačan rezultat.

Primjer:

```
if (A==0) A=B;
else
    A=A+4;
```

ako se A nalazi na 0(R3) a B na 0(R2), odgovarajući kôd je

```
        LW      R1, 0(R3)       ; čitanje A
        BNEZ    R1, L1          ; test A
        LW      R1, 0(R2)       ; if - then - slučaj
        J       L2              ; preskakanje else-a
L1:     ADDI    R1, R1, 4       ; else – slučaj
L2:     SW      0(R3), R1       ; upis A
```

Uz pretpostavku da se then-slučaj "gotovo uvijek" izvršava, i da je R14 "slobodan" registar, tada kompajler može spekulisati i generisati novi kôd

```
        LW      R1, 0(R3)               ; čitanje A
        LW      R14, 0(R2)              ; spekulativno čitanje B
        BEQZ    R1, L3
        ADDI    R14, R1, 4              ; else – slučaj
L3:     SW      0(R3), R1               ; nespekulativan upis A
```

Then - dio je potpuno spekulativan. *Else* - dio bi mogao biti kompajliran spekulativno sa uslovnim premiještanjem, ali ako je grananje jasno predvidivo i cijena promašaja nije visoka, to bi moglo usporiti kôd, jer se uvijek izvršavaju dvije dodatne instrukcije umjesto jednog grananja.

Nije ni važno da li je neka instrukcija spekulativna ili ne, to je korisno znati samo kada nastane greška u programu i desi se neprolazan izuzetak na normalnoj instrukciji - program se može završiti. Reimenovanje je često potrebno da bi se spriječile spekulativne instrukcije da unište "žive" podatke, pa je broj registara jedno od ograničenja u spekulisanju.

8.6.1. Spekulisanje korištenjem bita "otrova"

Korištenje bita otrova omogućava kompajlersko spekulisanje sa manjom promjenom uslova nastanka izuzetaka. Bit otrova je dodat svakom registru i drugi bit je dodat svakoj instrukciji da signalizira da li je ona spekulativna ili ne. Bit uz odredišni registar se postavlja kad god u njega vrši upis spekulativna instrukcija koja izaziva neprolazan izuzetak. Ako normalna instrukcija pokuša da koristi izvorišni registar sa postavljenim bitom otrova, ona izaziva grešku.

Prethodni primjer kompajliran sa spekulativnim instrukcijama i bitima otrova (označeno sa *) izgleda ovako (pretpostavka - registri R14 i R15 su "slobodni"):

```
        LW      R1, 0(R3)               ; čitanje A
        LW*     R14, 0(R2)              ; spekulativno čitanje B
        BEQZ    R1, L3
        ADDI    R14, R1, 4
    L3: SW      0(R3), R14          ; izuzetak za spekulativnu LW
```

Ako LW* generiše neprolazan izuzetak, bit otrova uz R14 će biti postavljen, pa kada nespekulativna SW pokuša koristiti R14 - izazvaće izuzetak.
Da bi OS bio u stanju sačuvati korisničke registre i sa postavljenim bitima otrova, neophodne su posebne instrukcije za čuvanje i resetovanje stanja ovih bita.

8.6.2. Spekulativne instrukcije sa reimenovanjem

Glavni nedostatak prethodne dvije metode je u tome što je potrebno uvođenje kopija da bi se riješilo reimenovanje registara i mogućnost nedostatka registara. Alternativa je premještanje instrukcija preko grananja, označavanje istih kao spekulativnih i omogućavanje reimenovanja i baferovanja u hardveru - slično Tomasulu. Ovako "pogurane" (eng. boosted) instrukcije se spekulativno izvršavaju zavisno od rezultata grananja. Kada se dođe do grananja, rezultati pogurane instrukcije iz pravilno predviđenog grananja se upisuju (eng. commit) u registre, inače se odbacuju. Guranje instrukcija preko više grananja je moguće, ali usložnjava upravljanje.
Prethodni primjer sa guranjem instrukcije preko grananja (označeno sa +) i predviđanjem da će se grananje desiti, izgleda ovako:

```
        LW    R1, 0(R3)          ; čitanje A
        LW+   R1, 0(R2)          ; pogurano čitanje B
        BEQZ  R1, L3
        ADDI  R1, R1, 4          ; else dio
L3:     SW    0(R3), R1          ; nespekulativan upis
```

Nisu potrebni dodatni registri, jer ako se grananje ne desi - rezultat spekulativnog čitanja se ne upisuje u R1. Rezultat pogurane instrukcije se ne piše u R1 do iza grananja, pa grananje koristi vrijednost R1 od prvog "nepoguranog" čitanja.

8.7. Hardversko spekulisanje

Hardversko spekulisanje kombinuje tri osnovne ideje:

1. dinamičko predviđanje grananja (za izbor instrukcija za izvršenje),
2. spekulaciju (za omogućavanje izvršenja instrukcija prije razrješenja upravljačkih zavisnosti) i
3. dinamičko raspoređivanje instrukcija (za raspoređivanje različitih kombinacija osnovnih blokova - sekvenci kôda).

Hardversko spekulisanje koristi dinamičku zavisnost podataka da izabere kada da izvrši instrukcije. Ovaj metod izvršenja programa se zove upravljanje tokom podataka (eng. data-flow execution) - operacije se izvršavaju čim njihovi operandi postanu raspoloživi.
Prednosti hardverskog u odnosu na softversko spekulisanje su:

1. da bi se povećao nivo spekulisanja, potrebno je razriješiti nejasnoće (dvosmislenosti) kod pristupa memoriji. Taj problem se naročito javlja kod cjelobrojnih programa koji koriste pokazivače. Hardversko rješenje ovog problema se postiže tehnikama viđenim kod Tomasulovog algoritma. Na taj način je moguće premještati instrukcije čitanja preko instrukcija pisanja u toku izvršenja programa.

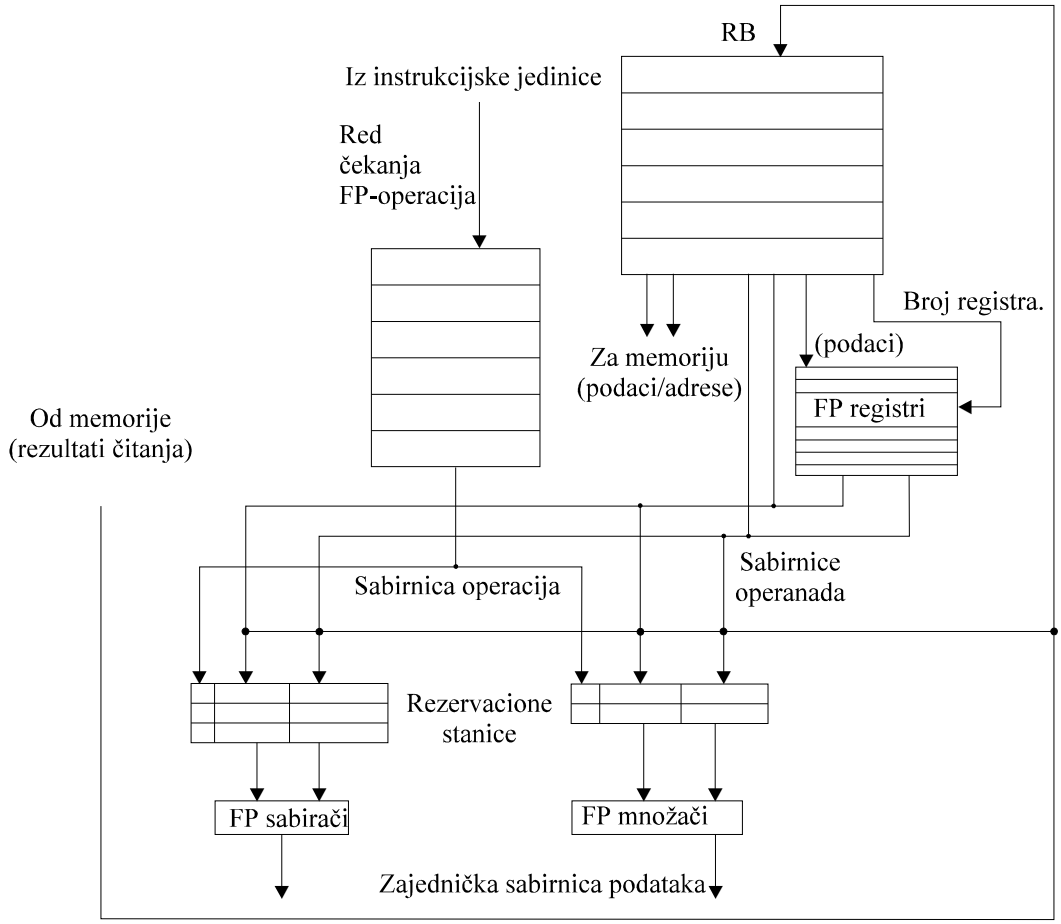
2. hardversko spekulisanje postiže bolje rezultate tamo gdje je dinamičko (hardversko) predviđanje grananja superiorno u odnosu na statičko.
3. hardversko spekulisanje održava potpuno preciznu obradu izuzetaka, čak i kod spekulisanih instrukcija.
4. hardversko spekulisanje ne zahtijeva nikakav prilagodni ili "knjigovodstveni" kod.
5. hardversko spekulisanje sa dinamičkim raspoređivanjem ne zahtijeva rekompajliranje kôda da bi se postigle optimalne performanse na različitim verzijama arhitektura.

Nasuprot ovim prednostima je glavni nedostatak - složenost neophodnog hardvera.
Komercijalni procesori firmi Intel i AMD, kombinuju spekulativno izvršavanje sa dinamičkim raspoređivanjem baziranim na Tomasulovom algoritmu.
Hardver za realizaciju Tomasulovog algoritma se može proširiti za podršku spekulisanju. Potrebno je razdvojiti prosljeđivanje rezultata među instrukcijama, koje je neophodno za razlikovanje spekulativno izvršenih instrukcija i njihovog stvarnog završetka. Na ovaj način je moguće izvršenje spekulativne instrukcije i prosljeđivanje njenog rezultata drugim instrukcijama, bez mogućnosti te instrukcije da izvrši bilo kakve promjene koje se ne mogu anulirati, sve do trenutka kada instrukcija prestane biti spekulativna. Tek tada ona upisuje novo stanje u skup registara ili memoriju. Time se dodaje još jedan korak u izvršenju instrukcije (eng. commit).
Osnovna ideja spekulisanja je da se omogući izvršenje instrukcija mimo reda, ali da se obezbijedi njihov konačni završetak originalnim redom. Kod jednostavne ogledne PS-e ovo se može postići (nakon svih izuzetaka detektovanih za datu instrukciju) pomjeranjem upisa na kraj PS-e. Da bi se uvelo spekulisanje, potrebno je razdvojiti izvršenje instrukcije od proglašenja njenih rezultata važećim, jer ova dva događaja mogu biti vremenski značajno razdvojeni. Dodavanje jedne faze u sekvencu izvršenja instrukcija, zahtijeva ne samo promjene na toj sekvenci, već dodatni hardver - skup bafera koji čuvaju rezultate instrukcija koji još nisu važeći. Ovaj bafer se zove bafer promjene redoslijeda - **RB** (eng. reorder buffer) i koristi se i za proslijeđivanje rezultata među spekulativnim instrukcijama.
RB pretstavlja dodatne virtuelne (programeru nevidljive) registre, slično rezervacionim stanicama kod Tomasulovog algoritma. Glavna razlika je u tome što, kod Tomasulovog algoritma, kada instrukcija upiše svoj rezultat, svaka kasnije pokrenuta instrukcija će naći taj rezultat u skupu registara. Kod spekulisanja, skup registara se ne ažurira sve do upisa rezultata (kada se zna da instrukcija nije više spekulativna - treba se izvršiti); tako RB snabdijeva instrukcije operandima u međuvremenu. Radi jednostavnosti, može se reći da RB funkcionalno zamijenjuju ulazne i izlazne bafere, za vezu sa memorijom, kod Tomasulovog algoritma.
Svaka linija (eng. entry) u RB-u se sastoji od tri polja:

1. tipa instrukcije,
2. polja odredišta i
3. polja vrijednosti.

Prvo polje pokazuje da li je instrukcija grananje (i nema odredišta rezultata), upis u memoriju (odredište je memorijska lokacija), ili registarska operacija (ALU ili čitanje iz memorije, čije je odredište registar). Drugo polje daje broj registra (za ALU operacije i čitanje iz memorije) ili adresu (za upis u memoriju) upisa rezultata. Treće polje sadrži vrijednost rezultata instrukcije, sve do njegovog upisa (commit-a).
Slika 8.4. prikazuje strukturu FP-dijela procesora sa RB-om. Iako je funkcija RS-a kod reimenovanja zamijenjena RB-om, još uvijek je neophodno baferovanje operacija (i operanada) od trenutka njihovog pokretanja do trenutka početka izvršenja. To će i dalje raditi RS-e. Pošto svaka instrukcija ima mjesto u RB-u dok se njeni rezultati ne upišu (u registre ili

memoriju), rezultat se označava brojem linije RB-a umjesto broja RS-e. Zato se taj podatak (broj RB-a) mora nalaziti i u RS-ama.



Slika 8.4. Osnovna struktura FP jedinice ogledne arhitekture sa Tomasulo-vim algoritmom, proširena da bi podržala spekulisanje. Glavne promjene su dodavanje RB-a i eliminisanje buffers čitanja i pisanja (njihovu ulogu je preuzeo RB). Ovaj mehanizam se može proširiti na višestruko pokretanje instrukcijja proširivanjem CDB da bi se omogućilo višestruko završavanje operacija po ciklusu.

Koraci u izvršenju FP-instrukcija (ideje su iste i za cjelobrojne) su sada:

1. **pokretanje** - uzimanje instrukcije iz FP-reda čekanja operacija. Pokretanje iste ako postoji prazna RS-a i prazno mjesto u RB-u, slanje operanada u RS ako su u registrima ili RB-u, ažuriranje kontrolnih struktura da proglase bafer zauzetim. Broj RB-a se šalje u RS-u, tako da se on može koristiti za otkrivanje rezultata kada se ovaj pojavi na zajedničkoj sabirnici podataka. Ako su sve RS-e pune ili nema mjesta u RB-u, nastaje zastoj u pokretanju instrukcija dok se oba resursa ne oslobode. Ovaj korak se zove i raspoređivanje (eng. dispatch), kod arhitektura sa dinamičkim raspoređivanjem.
2. **izvršenje** - ako jedan ili više operanada nije spremno, nadgleda se CDB čekajući na upis traženog podatka. Ovaj korak provjerava RAW hazarde. Kada su oba operanda raspoloživa u RS-ama, izvršava se operacija.
3. **upis rezultata u privremeni smještaj** - kada je rezultat raspoloživ, upisuje se na CDB (sa brojem RB-a iz doba pokretanja instrukcije) i sa nje u RB, kao i u sve RS-e koje čekaju taj rezultat. RS se označi kao slobodna.

4. **konačni upis rezultata** - kada instrukcija, osim grananja sa pogrešnim predviđanjem, dođe do kraja RB-a i njen rezultat je prisutan u baferu, vrši se upis rezultata u registar ili memorijsku lokaciju i uklanja se instrukcija iz RB-a. Kada grananje sa pogrešnim predviđanjem dođe na isto mjesto, indicira da je spekulisanje bilo pogrešno. RB se očisti i izvršenje se nastavlja sa korektnog mjesta (iza grananja).

Nakon konačnog upisa rezultata instrukcije, njeno mjesto u RB-u se proglašava slobodnim. Da bi se izbjeglo mijenjanje brojeva-oznaka u RB-u, on se realizuje kao kružni bafer - red čekanja, pa se redoslijed u njemu mijenja samo nakon završetka instrukcije.

Primjer: raniji primjer kod Tomasulovog algoritma uz trajanje sabiranja 2 ciklusa, množenja 10 i dijeljenja 40. Neka je data sekvenca instrukcija:

```
LD      F6, 34(R2)
LD      F2, 45(R3)
MULTD   F0, F2, F4
SUBD    F8, F6, F2
DIVD    F10, F0, F6
ADDD    F6, F8, F2
```

Stanje statusnih tabela u trenutku kada je MULTD spremna za konačan upis rezultata je dato na tabeli 8.3. Vidi se da, iako je SUBD izvršena, ne upisuje svoj rezultat dok to ne uradi MULTD. Sve oznake u Qj i Qk poljima, kao i u poljima statusa registara su zamjenjena brojevima linija RB-a, a odredišno polje označava broj linije RB-a koja je odredište rezultata.

	Reservacione stanice						
Ime	Zauzeto	Op	Vj	Vk	Qj	Qk	Dest
Sabir1	Ne						
Sabir2	Ne						
Sabir3	Ne						
Množ1	Ne	MULT	Mem[45+Regs[R3]]	Regs[F4]			#3
Množ2	Da	DIV		Mem[34+Regs[R2]]	#3		#5

	RB				
Ulaz	Zauzeto	Instrukcija	Stanje	Odredište	Vrijednost
1	No	LD F6, 34(R2)	Konačni upis	F6	Mem[34+Regs[R2]]
2	No	LD F2, 45(R3)	Konačni upis	F2	Mem[45+Regs[R3]]
3	Yes	MULTD F0, F2, F4	Upis rezultata	F0	#2 × Regs[F4]
4	Yes	SUBD F8, F6, F2	Upis rezultata	F8	#1 - #2
5	Yes	DIVD F10, F0, F6	Izvršenje	F10	
6	Yes	ADDD F6, F8, F2	Upis rezultata	F6	#4 + #2

	Status FP registara								
Polje	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	3			6	4	5			
Zauzeto	Da	Ne	Ne	Da	Da	Da	Ne	...	Ne

Tabela 8.3. Samo dvije LD instrukcije su konačno upisale rezultate, iako je više drugih izvršeno. SUBD i ADDD se neće završiti dok se MULTD ne završi, iako su rezultati instrukcija raspoloživi i mogu biti izvorišta za druge instrukcije. Kolona vrijednost daje vrijednost koju drži, format #X se odnosi na vrijednost polja X rb-a.

Ovaj primjer ilustruje glavnu razliku između spekulisanja i dinamičkog raspoređivanja. U

ovom slučaju se konačan upis rezultata vrši originalnim redom pokretanja instrukcija.

Primjer: raniji primjer kod Tomasulovog algoritma. Neka je data sekvenca instrukcija:

```
Loop: LD      F0, 0(R1)
      MULTD   F4, F0, F2
      SD      0(R1), F4
      SUBI    R1, R1, #8
      BNEZ    R1, Loop      ; grananje ako R1 nije 0
```

Neka su pokrenute sve instrukcije u petlji dva puta, i da su LD i MULTD iz prve iteracije završene i njihovi rezultati konačno upisani, a sve ostale instrukcije su izvršene. Kod dinamičkog raspoređivanja i za FP i cjelobrojne FJ-e, upis u memoriju bi čekao u RB-u na efektivnu adresu (R1) i vrijednost (F4). Pošto se razmatra samo FP dio, neka je efektivna adresa poznata u trenutku pokretanja upisa u memoriju.
Rezultat je dat na tabeli 8.4.

	Rezervacione stanice						
Ime	Zauzeto	Op	Vj	Vk	Qj	Qk	Dest
Množ1	No	MULT	Mem[0+Regs[R1]]	Regs[F2]			#2
Množ2	No	MULT	Mem[0+Regs[R1]]	Regs[R2]			#7

	RB				
Ulaz	Zauzeto	Instrukcija	Stanje	Odredište	Vrijednost
1	Ne	LD F0, 0(R1)	Konačni upis	F0	Mem[0+Regs[R1]]
2	Ne	MULTD F4, F0, F2	Konačni upis	F4	Regs[F0]×Regs[F2]
3	Da	SD 0(R1), F4	Upis rezultata	0+Regs[R1]	#2
4	Da	SUBI R1, R1, #8	Upis rezultata	R1	R1 - 8
5	Da	BNEZ R1, Loop	Upis rezultata		
6	Da	LD F0, 0(R1)	Upis rezultata	F0	Mem[#4]
7	Da	MULTD F4, F0, F2	Upis rezultata	F4	#6 × Regs[F2]
8	Da	SD 0(R1), F4	Upis rezultata	0+Regs[R1]	#7
9	Da	SUBI R1, R1, #8	Upis rezultata	R1	#4 - 8
10	Da	BNEZ R1, Loop	Upis rezultata		

	Status FP registara								
Polje	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	6		7						
Zauzeto	Da	Ne	Da	Ne	Ne	Ne	Ne	...	Ne

Tabela 8.4. Samo LD i MULTD instrukcije su konačno upisale rezultate, iako su sve druge izvršene.
Preostale instrukcije će se završiti što je prije moguće.

Procesor može jednostavno odbaciti sve spekulativne rezultate kada se pokaže da je grananje pogrešno predviđeno. Neka se BNEZ grananje nije desilo prvi put. Instrukcije prije grananja će se završiti kada svaka dođe do kraja RB-a. Kada grananje dođe do kraja RB-a, bafer se čisti i počinje dobavljanje instrukcija iz druge grane.
Tabela 8.5. pokazuje korake izvršenja instrukcije, kao i uslove za prelazak u naredni korak.

Status instrukcije	Čekanje na	Akcija ili knjigovodstvo

Pokretanje	Stanica i RB raspoloživi	if (Register[’S1’].Busy)
		{RS[r].Qj←Register[’S1’].Reorder}
		else {RS[r].Vj←Regs[S1]; RS[r].Qj←0};
		if (Register[’S2’].Busy)
		{RS[r].Qk←Register[‘S2’].Reorder}
		else {RS[r].Vk←Regs[S2]; RS[r].Qk←0};
		RS[r].Busy←yes; RS[r].Dest←b;
		Register[’D’].Reorder=b; Register[’D’].Busy=true;
		Reorder[b].Dest←’D’;

Izvršenje	(RS[r].Qj=0) and (RS[r].Qk=0)	None - operands are in Vj and Vk
Upis rezultata	Izvršenje obavljeno u r sa baferom broj b (=RS[r].Dest) i CDB raspoloživom, vrijednost je result	∀x (if (RS[x].Qj=b) {RS[x].Vj←result; RS[x].Qj←0}); ∀x (if (RS[x].Qk=b) {RS[x].Vk←result; RS[x].Qk←0}); RS[r].Busy←No; Reorder[b].Value←result;
Konačni upis rezultata	Instrukcija je na vrhu-čelu RB-a (ulaz h) i završila je upis rezultata (prethodni korak)	if (Reorder[h].Instruction=Branch) and (branch is mispredicted) {clear reorder buffer and Register status; restart instruction fetch at successor;} else if (Reorder[h].Instruction=Store) {Mem[Reorder[h].Dest]←Reorder[h].Value;} else {Regs[Reorder[h].Dest]←Reorder[h].Value;} Register[Reorder[h]].Busy←No; Reorder[h].Busy←No;

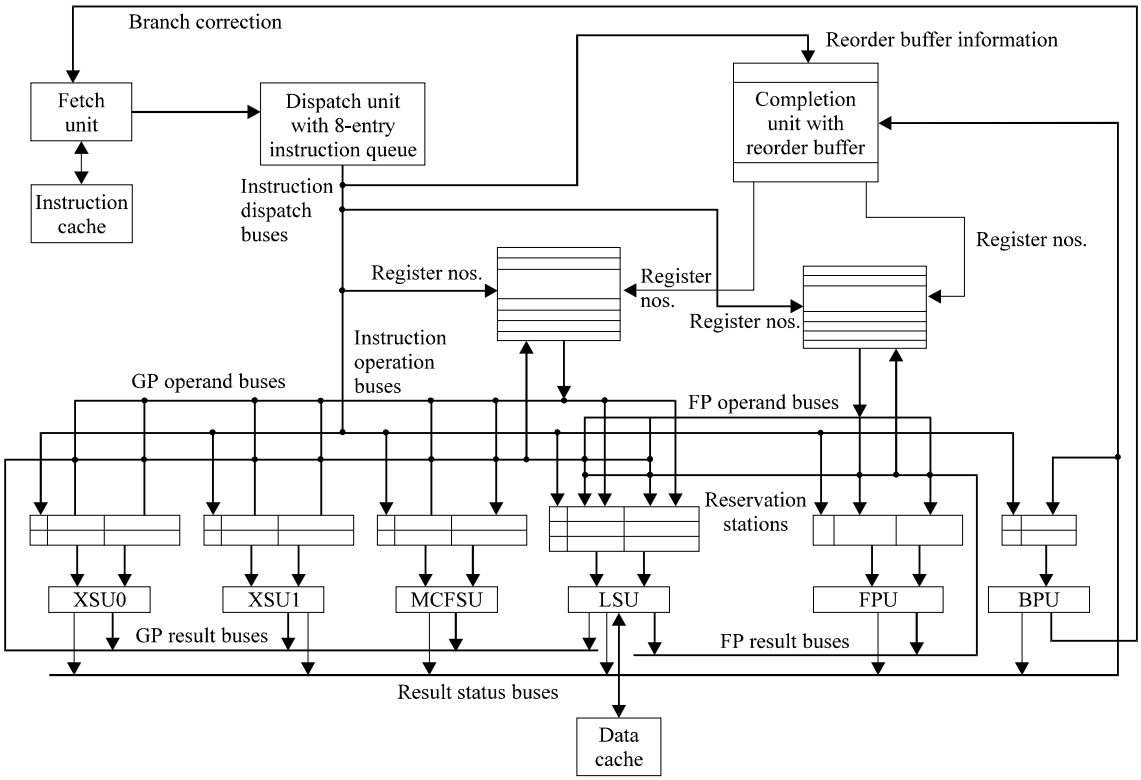
Tabela 8.5. Koraci u algoritmu i šta je potrebno u svakom od njih. Za instrukciju koja se pokreće, **D** je odredište, **S1** i **S2** su izvorišta, **r** je alocirana RS, i **b** je dodijeljeni ulaz RB-a. **RS** je struktura podataka RS-e. Vrijednost koju vraća RS se zove **result**. **Register** je registarska struktura podatala, **Regs** pretstavlja stvarne registre, dok je **Reorder** struktura podataka RB-a. **’Ri’** označava ime registra **Ri,** a ne njegovu vrijednost.

Ova tehnika je naročito interesantna kod procesora koji pokreću (i, u tom slučju, završavaju) više instrukcija po ciklusu. Problemi su tada:

1. koje instrukcije se mogu pokrenuti,
2. kako izvršiti reimenovanje u kratkom vremenskom intervalu (ciklusu sata),
3. nadgledanje više CDB-ova (u jednom ciklusu).

Da li treba podržati spekulisanje prvenstveno u hardveru ili softveru, ne može se sa sigurnošću tvrditi. U oba slučaja, a kao što je i ranije rečeno, ne može se postići viši nivo paralelizma na nivou instrukcija od onog “ugrađenog” u kôd koji se izvršava.

Slika 8.5. prikazuje strukturu komercijalnog procesora PowerPC 620.



Slika 8.5. PowerPC 620 ima 6 različitih funkcionalnih jedinica, svaka ima svoju RS i 16-ulazni RB, sadržan u jedinici za dovršenje instrukcije (instruction completion unit). Reimenovanje je implementirano i za cjelobrojne i FP-registre, a dodatni registri za reimenovanje su dio odgovarajućih skupova registara.

U tabeli 8.6. su date karakteristike nekih komercijalnih procesora .

					Mogućnosti pokretanja instrukcija					
Procesor	Godina pojave	Frekv. sata (MHz)	Struktura pokretanja instrukcija	Raspore đivanje	Maksi malno	Čitanje / pisanje	Int. ALU	FP	Grana-nje	SPEC
IBM Power-1	1991	66	Dinamičko	Statičko	4	1	1	1	1	60 int 80 FP
HP 7100	1992	100	Statičko	Statičko	2	1	1	1	1	80 int 150 FP
DEC Alpha 21064	1992	150	Dinamičko	Statičko	2	1	1	1	1	100 int 150 FP
Super-SPARC	1993	50	Dinamičko	Statičko	3	1	1	1	1	75 int 85 FP
IBM Power-2	1994	67	Dinamičko	Statičko	6	2	2	2	2	95 int 270 FP
MIPS TFP	1994	75	Dinamičko	Statičko	4	2	2	2	1	100 int 310 FP
Intel Pentium	1994	66	Dinamičko	Statičko	2	2	2	1	1	65 int 65 FP
DEC Alpha 21164	1995	300	Statičko	Statičko	4	2	2	2	1	330 int 500 FP
Sun Ultra-SPARC	1995	167	Dinamičko	Statičko	4	1	1	1	1	275 int 305 FP
Intel P6	1995	150	Dinamičko	Dinam.	3	1	2	1	1	>200 int
AMD K5	1995	100	Dinamičko	Dinam.						130
HaL R1	1995	154	Dinamičko	Dinam.	4	1	2	1	1	255 int 330 FP
PowerPC 620	1995	133	Dinamičko	Dinam.	4	1		1	1	225 int 300 FP
MIPS R10000	1995	200	Dinamičko	Dinam.	4	1	2	2	1	300 int 600 FP
HP 8000	1996	200	Dinamičko	Statičko	4	2	2	2	1	>360 int >550 FP

Tabela 8.6. Neki komercijalni procesori visokih performansi i njihove karakteristike.

9. Superskalarni i VLIW procesori

U prethodnim poglavljima je bilo govora o metodama za odklanjanja upravljačkih i hazarda/zastoja podataka, sa ciljem da se postigne, u idealnom slučaju, izvršenje jedne instrukcije u jednom ciklusu sata (CPI=1). Da bi se ovaj parametar i dalje smanjio, neophodno je pokretanje više od jedne instrukcije po ciklusu. Takvi procesori se dijele na:

- superskalarne i
- VLIW (eng. Very Long Instruction Word).

Superskalarni procesori pokreću različit broj instrukcija po ciklusu i koriste statičko ili dinamičko raspoređivanje instrukcija. VLIW procesori pokreću određen broj instrukcija formatiranih kao jedna velika instrukcija ili kao paket instrukcija, a koriste statičko raspoređivanje instrukcija.

Za poređenje ova dva pristupa, poslužiće jedan od ranijih primjera - dodavanje skalara nizu u memoriji:

```
Loop:   LD      F0, 0(R1)     ; F0 = element niza
        ADDD    F4, F0, F2    ; dodavanje skalara iz F2
        SD      0(R1), F4     ; smještanje rezultata
        SUBI    R1, R1, #8    ; dekrement pokazivača na sljedeću DW
        BNEZ    R1, Loop      ; grananje ako je R1!=0
```

9.1. Superskalarna verzija ogledne arhitekture

Kod tipičnog superskalarnog procesora, instrukcije koje se istovremeno pokreću moraju biti nezavisne i zadovoljiti određene preduslove - npr. dozvoljen je samo jedan pristup memoriji po ciklusu. U protivnom samo njoj prethodne instrukcije će biti pokrenute u tom ciklusu - zato varira broj pokrenutih instrukcija po ciklusu. Nasuprot tome, kod VLIW procesora odgovornost je na kompajleru da kreira pakete instrukcija koje se mogu istovremeno pokrenuti - na to hardver ne može naknadno uticati.

Neka superskalarna ogledna arhitektura može pokrenuti dvije instrukcije po ciklusu. Jedna od njih može biti čitanje ili pisanje u memoriju, grananje ili cjelobrojna ALU operacija, a druga bilo koja FP-operacija. Ovo je znatno jednostavnije realizovati nego pokretanje bilo koje dvije instrukcije istovremeno.

Pokretanje i dekodiranje dvije instrukcije po ciklusu zahtijeva dobavljanje 64 bita instrukcija. Radi jadnostavnosti, neka su instrukcije uparene i 64-bitno poravnate u memoriji, sa cjelobrojnom na prvom mjestu. Druga instrukcija se može pokrenuti samo ako može i prva - hardver o tome dinamički odlučuje. Tabela 9.1. prikazuje izgled instrukcija u PS-i.

Tip instrukcije	Segmenti PS-e							
Cjelobr. instrukcija	IF	ID	EX	MEM	WB			
FP instrukcija	IF	ID	EX	MEM	WB			
Cjelobr. instrukcija		IF	ID	EX	MEM	WB		
FP instrukcija		IF	ID	EX	MEM	WB		
Cjelobr. instrukcija			IF	ID	EX	MEM	WB	
FP instrukcija			IF	ID	EX	MEM	WB	
Cjelobr. instrukcija				IF	ID	EX	MEM	WB

FP instrukcija	IF	ID	EX	MEM	WB

Tabela 9.1. Superskalarna PS-a u radu. Cjelobrojne i FP instrukcije se pokreću istovremeno, i svaka se izvršava svojom brzinom u PS-i. Ovako će se ubrzati izvršenje programâ sa značajnom količinom FP operacija.

Na ovaj način, značajno je povećan broj pokretanja FP-operacija, pa je neophodno imati ili protočne FP-jedinice ili više nezavisnih FP-jedinica. U protivnom, dobit od višestrukog pokretanja instrukcija bi se smanjila zbog uskog grla u FP-dijelu arhitekture. Paralelnim pokretanjem cjelobrojne i FP-operacije, minimizirana je potreba za dodatim hardverom - koriste se odvojeni skupovi instrukcija i FJ-e.

Restrikcije pri pokretanju se mogu realizovati poređenjem samo kodova operacija. Jedina komplikacija je kada cjelobrojna instrukcija vrši čitanje/pisanje/premiještanje FP-registara - nastaje strukturni hazard, konflikt nad portovima FP-registara. Rješenja su - ne pakovati takve dvije instrukcije zajedno ili realizovati FP registre sa dva porta - jednim za čitanje i jednim za pisanje.

Postoji još jedna poteškoća koja može ograničiti učinak superskalarne PS-e. Kod jednostavne ogledne PS-e, čitanje iz memorije je izazivalo zastoj od jednog ciklusa. Kod superskalarne PS-e to znači da se rezultat čitanja ne može koristiti ni u istom ni u sljedećem ciklusu. Tako tri sljedeće instrukcije ne mogu koristiti rezultat čitanja iz memorije, bez zastoja. Kašnjenje kod grananja utiče na zastoj tri instrukcije (a ne jednu kao ranije) jer se instrukcija grananja pakuje kao prva u paru. Zato superskalarne arhitekture zahtijevaju naprednije tehnike raspoređivanja instrukcija i njihovo složenije dekodiranje.

Primjer: ranije navedeni kod, “odmotan” za superskalarnu ogledni PS-u, uz ranije utvrđena kašnjenja (tabela 5.1) je dat u tabeli 9.2. Odmotano je pet kopija petlje i raspoređeno bez zastoja. Izvršava se za 12 ciklusa po iteraciji, ili 2,4 ciklusa po elementu (tijelu petlje) dok je kod klasične ogledne PS-e to 3,5. U ovom primjeru, performanse su ograničene ravnotežom između cjelobrojnih i FP-operacija - teško je FP PS-u držati popunjenom. Odmotavanje petlje je donijelo ubrzanje od 6 na 3,5 ciklusa (faktor 1,7) dok je superskalarno izvršenje donijelo dodatni faktor ubrzanja od 1,5.

Superskalarni procesor će pokrenuti dvije instrukcije ako je prva cjelobrojna a druga FP-operacija. Ako to nije slučaj (što je lako otkriti), instrukcije se pokreću sekvencijalno. To ukazuje na dvije najvažnije prednosti superskalarnih arhitektura u odnosu na VLIW-arhitekture:

- procesor određuje da li se sljedeća instrukcija može pokrenuti i
- program se može izvršiti i bez optimalnog raspoređivanja instrukcija (sporo ali moguće) - što se može prevazići dinamičkim raspoređivanjem.

	Cjelobrojna instrukcija	FP instrukcija	Ciklus sata
LOOP:	LD F0, 0(R1)		1
	LD F6, -8(R1)		2
	LD F10, -16(R1)	ADDD F4, F0, F2	3
	LD F14, -24(R1)	ADDD F8, F6, F2	4
	LD F18, -32(R1)	ADDD F12, F10, F2	5
	SD 0(R1), F4	ADDD F16, F14, F2	6
	SD -8(R1), F8	ADDD F20, F18, F2	7
	SD -16(R1), F12		8
	SD -24(R1), F16		9
	SUBI R1, R1, #40		10
	BNEZ R1, LOOP		11
	SD -32(R1), F20		12

Tabela 9.2. Odmotana i preraspodijeljena sekvenca instrukcija na superscalarnoj oglednoj PS-i.

9.2. Višestruko pokretanje instrukcija sa dinamičkim raspoređivanjem

Pokretanje više instrukcija istovremeno se može primijeniti i kod procesora sa dinamičkim raspoređivanjem - bilo sa semaforom ili Tomasulovim algoritmom. Slanje instrukcija RS-ma mimo reda bi učinilo praćenje (knjigovodstvo) vrlo složenim. Korištenjem odvojenih struktura podataka za cjelobrojne i FP-registre, mogu se istovremeno pokrenuti jedna FP i jedna cjelobrojna instrukcija u njihove odgovarajuće RS-e, dok god one ne pristupaju istom skupu registara.

Ovaj pristup zabranjuje pokretanje dvije zavisne instrukcije u jednom ciklusu. Ako hardversko raspoređivanje nije u stanju da razriješi ovaj problem, onda se sve svodi na kompajlersko - statičko raspoređivanje.

Jedno rješenje je realizovati segment pokretanja istrukcija kao PS-u, tako da radi duplo brže od osnovnog takta. Tako je moguće ažurirati tabele prije pokretanja sljedeće instrukcije - tada se dvije instrukcije mogu početi izvršavati u istom ciklusu.

Drugi pristup počiva na ograničenju da će samo FP-čitanja iz memorije i premještanja iz cjelobrojnih registara opšte namjene u FP-registre izazivati zavisnosti među zajedno pokrenutim instrukcijama. Mogućnost pokretanja bez ovog ograničenja bi izazvalo druge zavisnosti koje bi trebalo rješavati.

Potreba za RS-ma kod čitanja iz memorije i premiještanjima među registrima se može riješiti pomoću redova čekanja (eng. queues) za rezultate tih operacija. Oni se mogu koristiti za rano pokretanje upisa u memoriju i njihovo čekanje na operande, kao kod Tomasulovog algoritma.

Kako je dinamičko raspoređivanje najefikasnije kod premiještanja podataka između registara i memorije, a statičko među registrima, može se koristiti statičko raspoređivanje za eliminisanje RS-a a osloniti se na redove čekanja kod load/store operacija.

Dinamičko raspoređivanje operacija čitanja/pisanja može rezultirati u promjeni njihovog redoslijeda. To može dovesti do zavisnosti podataka u memoriji i zahtijeva dodatne mehanizme (hardver) za otkrivanje takvih hazarda. To se može riješiti kao kod Tomasula - dinamičkom provjerom da li je adresa izvorišta u memoriji ista kao adresa odredišta neke od nedovršenih instrukcija. Ako se otkrije takav konflikt, čitanje se zadržava dok se pisanje ne završi.

Primjer: neka je segment za pokretanje instrukcija realizovan kao PS, pa je moguće pokrenuti i dvije zavisne instrukcije koje koriste odvojene FJ-e, uz uslov da je cjelobrojna prva. Neka je vrijeme izvršenja (broj ciklusa kašnjenja po instr.) svake instrukcije isto. Neka pokretanje i pisanje rezultata traje po jedan ciklus i koristi se hardver za dinamičko predviđanje grananja.

Ranije navedena sekvenca koda (petlja) će se dinamički “odmotati” i, kada god je to moguće, instrukcije će se pokrenuti u parovima.

Tabela 9.3. daje rezultat - petlja se odvija u 4=7/n ciklusa po rezultatu za n iteracija. Za veliko n to se približava vrijednosti od 4 ciklusa po rezultatu.

Broj dvostrukih pokretanja je mali jer postoji samo jedna FP-operacija u iteraciji petlje. Daljnje “odmotavanje” petlje i smanjanje broja instrukcija za održavanje petlje bi povećalo relativni broj dvostrukih pokretanja. Proširenje zajedničke sabirnice podataka (CDB-a) bi

omogućilo pokretanje više cjelobrojnih operacija po ciklusu i značajnije povećanje performansi.

Broj iteracije	Instrukcije	Pokretanje u ciklusu broj	Izvršenje u ciklusu broj	Upis rezultata u ciklusu broj
1	LD F0, 0(R1)	1	2	4
1	ADDD F4, F0, F2	1	5	8
1	SD 0(R1), F4	2	9	
1	SUBI R1, R1, #8	3	4	5
1	BNEZ R1, LOOP	4	5	
2	LD F0, 0(R1)	5	6	8
2	ADDD F4, F0, F2	5	9	12
2	SD 0(R1), F4	6	13	
2	SUBI R1, R1, #8	7	8	9
2	BNEZ R1, LOOP	8	9	

Tabela 9.3. Vrijeme pokretanja, izvršenja, i upisa rezultata kod PS-e sa Tomasulo-algoritmom i dvostrukim pokretanjem instrukcija. Upis rezultata se ne odnosi na pisanja u memoriju ili grananja, jer se tada ne upisuje u registre. Podrazumijeva se šira CDB i više portova za upis u registre, koji su potrebni u 8 ciklusu ovog primjera.

9.3. VLIW pristup

Korištenje VLIW može smanjiti složenost hardvera potrebnog za realizaciju procesora sa višestrukim pokretanjem instrukcija i dinamičkim raspoređivanjem instrukcija.

Superskalarni procesor koji pokreće dvije instrukcije u jednom ciklusu zahtijeva poređenje dva koda operacija i šest specifikatora registara, kao i dinamičko određivanje da li je moguće pokretanje jedne ili dvije instrukcije. Sa povećanjem broja instrukcija koje se žele pokrenuti u jednom ciklusu, postaje složeno određivanje mogućnosti njihovog pokretanja bez poznavanja njihovog originalnog redoslijeda dobavljanja i mogućih međuzavisnosti.

Alternativa je VLIW - korištenje više nezavisnih FJ. Umjesto pokušavanja da pokrene više nezavisnih instrukcija, VLIW pakuje više operacija u jednu dugu instrukciju. Kako je zadatak određivanja instrukcija za pokretanje dat kompajleru, hardver za taj dio posla nije potreban.

Neka VLIW instrukcija može sadržati dvije cjelobrojne operacije, dvije FP-operacije, dva pristupa memoriji i grananje. Svaka bi FJ imala svoje polje u instrukciji (16 do 24 bita) što bi dovelo do ukupne dužine od 112 do 168 bita. Da bi se zaposlile sve FJ-e, mora postojati dovoljan nivo paralelizma u kodu. Taj paralelizam se postiže “odmotavanjem” petlji i raspoređivanjem instrukcija na nivou iznad osnovnih segmenata koda.

Primjer: neka je dat VLIW procesor koji može pokrenuti dva pristupa memoriji, dvije FP-operacije i jednu cjelobrojnu operaciju ili grananje u svakom ciklusu sata. Izvršenje ranije definisanog koda je dato u tabeli 9.4. Petlja je “odmotana” sedam puta, eliminisani su svi zastoji i izvršava se u 9 ciklusa. Tako je postignuto dobijanje 7 rezultata u 9 ciklusa (1.28 ciklusa po rezultatu).

Pristup memoriji 1	Pristup memoriji 2	FP operacija 1	FP operacija 2	Cjelobrojna operacija/grananje
LD F0, 0(R1)	LD F6, -8(R1)			
LD F10, -16(R1)	LD F14, -24(R1)			
LD F18, -32(R1)	LD F22, -40(R1)	ADDD F4, F0, F2	ADDD F8, F6, F2	
LD F26, -48(R1)		ADDD F12, F10, F2	ADDD F16, F14, F2	
		ADDD F20, F18, F2	ADDD F24, F22, F2	

SD 0(R1), F4	SD -8(R1), F8	ADDD F28, F26, F2	
SD -16(R1), F12	SD -24(R1), F16		
SD -32(R1), F20	SD -40(R1), F24		SUBI R1, R1, #56
SD -0(R1), F28			BNEZ R1, LOOP

Tabela 9.4. VLIW instrukcije iz petlje koje zamijenjuju odmotanu sekvencu. Ovom kodu treba 9 ciklusa pod uslovom da nema kašnjenja zbog grananja. Učestanost pokretanja je 23 operacije u 9 ciklusa sata, ili 2.5 po ciklusu. Efikasnost popunjavanja resursa je oko 60%. Ovo zahtijeva korištenje većeg broja registara u odnosu na broj koji bi ogledna arhitektura koristila u ovoj petlji. VLIW zahtijeva najmanje 8 FP registara, dok bi osnovni ogledni procesor mogao koristiti od 2 do 5, zavisno od odmotavanja i preraspoređivanja. U primjeru superskalara (slika 4.27), bilo je potrebno 6 registara.

9.4. Ograničenja procesora sa višestrukim pokretanjem instrukcija

Poteškoće u povećanju broja instrukcija koje se pokreću u svakom ciklusu se mogu podijeliti na:
- inherentna ograničenja raspoloživim paralelizmom na nivou instrukcija (ILP) u programu,
- problemi u realizaciji neophodnog hardvera za podršku, i
- posebna ograničenja vezana za superskalarne ili VLIW strukture.

Ograničenja raspoloživog ILP-a su najjednostavnija i najosnovnija. Ako, kod statičkog raspoređivanja, petlje nisu mnogo puta "odmotane", neće biti dovoljno operacija raspoloživih za zapošljavanje svih funkcionalnih jedinica. Na prvi pogled, pet instrukcija koje se mogu izvršavati paralelno, mogu držati spomenutu VLIW strukturu potpuno zaposlenom. Međutim, više funkcionalnih jedinica - memorija, grananja i FP-jedinica - mogu biti realizovano kao PS-e i imati višeciklusno kašnjenje, zahtijevajući veliki broj operacija koje se mogu izvršavati paralelno, da bi se izbjegli zastoji. Ako FP-PS ima kašnjenje od 5 ciklusa, za zapošljavanje dviju takvih funkcionalnih jedinica je potrebno 10 FP-operacija nezavisnih od već pokrenutih FP-operacija. U opštem slučaju, potrebno je naći broj nezavisnih operacija jednak prosječno broju (dubini) segmenata PS-a pomnoženim sa brojem funkcionalnih jedinica. To znači 15 do 20 operacija treba da se zaposli struktura sa 5 funkcionalnih jedinica.

Dodatni hardver je neophodan za realizaciju pokretanja i izvršenja višestrukih operacija svakog takta. Dupliciranje FP i cjelobrojnih funkcionalnih jedinica nije problem - troškovi rastu linearno. Međutim, neophodno je veliko povećanje propusnosti memorijskog podsistema i skupa registara. Čak i sa odvojenim cjelobrojnim i FP-skupom registara, VLIW procesor bi zahtijevao 6 portova za čitanje (po dva za svaku load/store i dva za cjelobrojni dio) i 3 porta za pisanje (po jedan za svaki ne-FP funkcionalnu jedinicu) za skup cjelobrojnih registara i 6 portova za čitanje (po dva za svaku load/store i FP funkcionalnu jedinicu) i 4 porta za pisanje (po jedan za svaku load/store ili FP funkcionalnu jedinicu) za skup FP-registara. Sve to zahtijeva trošenje veće površine silicijuma za realizaciju skupa registara, a utiče i na moguće brzine signala sata. VLIW sa 5 FJ treba imati dva porta za prenos podataka sa memorijom, što je znatno skuplje od registarskih portova. Složenost i vrijeme pristupa višeportnim memorijskim strukturama su najozbiljnija hardverska ograničenja kod svakog procesora sa višestrukim pokretanjem instrukcija, i kod superskalarnih i kod VLIW.

Najviše dodatnog hardvera je potrebno za realizaciju superskalarnih procesora sa dinamičkim raspoređivanjem instrukcija. Takvi dizajni su i najsloženiji, najteže postižu visoke frekvencije taktnih signala i njihov dizajn se teško verifikuje. S druge strane VLIW procesori zahtijevaju malo ili ni malo dodatnog hardvera, jer sav posao odrađuju kompajleri. Većina komercijalnih

procesora je razultat kompromisa između ove dvije krajnosti.

Na kraju, postoje problemi koji se odnose posebno na superskalarne ili VLIW strukture. Glavni problem kod superskalarnih je, kao što je već rečeno, logika za pokretanje instrukcija. Kod VLIW postoje i tehnički i logistički problemi. Tehnički su povećanje koda (odmotavanje petlji) i problem zapošljavanja funkcionalnih jedinica. U tabeli 5.25 samo oko 60% funkcionalnih jedinica se koristi - skoro pola instrukcija je prazno! S druge strane, zastoj u svakoj funkcionalnoj jedinici izaziva zastoj čitavog procesora, jer sve jedinice moraju biti sinhronizovane (npr. promašaj u kešu). Glavni logistički problem kod VLIW je binarna kompatibilnost koda (čak i kod procesora sa istim skupom instrukcija). Svaka promjena unutrašnje strukture zahtijeva rekompilaciju kôda.
Glavni izazov kod procesora sa višestrukim pokretanjem instrukcija je traženje i iskorištavanje ILP-a.

10. Memorijska hijerarhija

U prethodnim poglavljima bilo je govora o dizajnu procesora, glavne komponente svakog računara. Osnovni cilj je bio postizanje viših performansi – izvršenje što više instrukcija u što manjem broju ciklusa sata. To je povećalo zahtjeve za poboljšanjem performansi memorije, jer je potrebno dobaviti, u najjednostavnijem slučaju, jednu instrukciju svakog ciklusa sata (ako se želi iskoristiti raspoloživi paralelizam na nivou instrukcija, tada je potrebno dobavljati više instrukcija istovremeno!). Porasli su i zahtjevi za prenosom podataka iz memorije podataka ka procesoru i obratno.

Do sada je memorija posmatrana kao horizontalna uniformna struktura, jednostavan niz lokacija – riječi, za čije čitanje je potrebna adresa, a za pisanje adresa i podatak. Smatralo se da pristup svakoj lokaciji traje isti vremenski period. Od sredine 1980-tih godina, tehnologija izrade memorijskih komponenti ne omogućava tako organizovanoj memoriji da odgovori zahtjevima sve bržih procesora. Zato je neophodna složenija memorijska struktura koja se naziva memorijskom hijerarhijom.

Prema tehnologiji izrade, savremene memorijske komponente se mogu podijeliti na:

1. poluprovodničke (registri, SRAM, DRAM, FLASH),
2. magnetne (fleksibilni diskovi FDD, tvrdi diskovi HDD) i
3. optičke (CD, DVD).

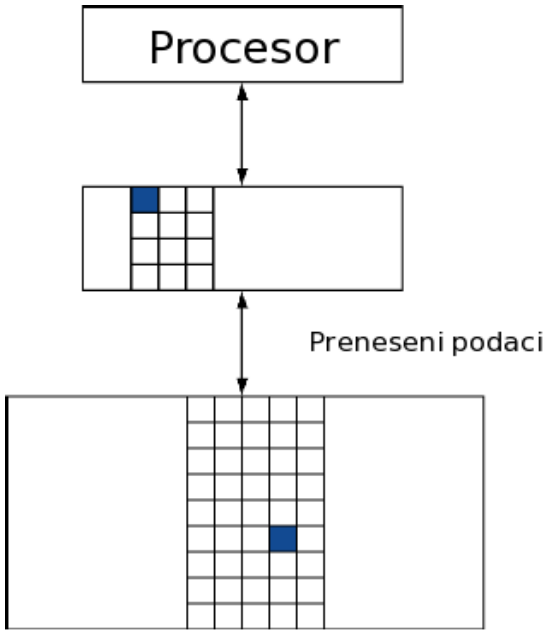
Poluprovodničke nemaju pokretnih dijelova (eng. solid state) i vrijeme pristupa svim lokacijama je isto (eng. radnom access, mada nema ništa slučajno u pristupu memoriji i instrukcija i podataka!).

Kod magnetnih i optičkih memorija, magnetni i optički mediji su pokretni. Magnetni pohranjuju podatke (nule i jedinice) različitom polarizacijom namagnetisavanja medija, dok se kod optičkih upis vrši uklanjanjem ili ostavljanjem reflektujućeg sloja na diskovima. Zbog okretanja medija, vrijeme pristupa svakoj lokaciji na diskovima nije isto. Da bi se mehanizam za čitanje i pisanje namjestio precizno iznad tražene lokacije na disku, statistički je potrebno čekati onoliko koliko disku treba da napravi pola kruga (najmanje nekoliko milisekundi). Do tog trenutka, pristup se može smatrati slučajnim, iako vrijeme pomicanja glave do svakog mjesta (staze) na disku nije isto. Od tog trenutka počinje sekvencijalni upis ili čitanje podataka sa izabrane staze.

Kako poluprovodničke (elektroničke) memorije nemaju pokretnih dijelova, one su mnogo brže. S druge strane, kapacitet magnetnih i optičkih memorija je neuporedivo (za red veličine) veći nego kod poluprovodničkih, a cijena po jedinici informacije je neuporedivo niža. Naravno postoje razlike u kapacitetu i cijeni i unutar pomenutih grupa memorijskih komponenti. Registri su brži i skuplji od statičke memorije, koja je brža i skuplja od dinamičke memorije, itd. Slični odnosi važe za HDD i FDD, kao i DVD i CD. Očigledno je neophodno voditi računa o kompromisima prilikom projektovanja memorijskog sistema, obzirom za velike razlike u brzini, kapacitetu i cijeni navedenih komponenti. Kao i u drugim slučajevima, kada ljudi ne mogu da “prevare” zakone fizike, u pomoć pozovu statistiku! Pomoću nje se lakše grade iluzije!

Osnovna ideja hijerarhijske strukture memorije jeste korištenje više različitih tipova memorijskih komponenti, sa ciljem da se iskoriste najbolje osobine svake od njih. Logično je da se najbrža memorija smjesti najbliže centralnoj procesnoj jedinici (CPU), slika 10.1. Sljedeća, do nje, će biti nešto sporija, pa još sporija itd. Memorijska struktura najbliža CPU-u, po pravilu je i najmanjeg kapaciteta i najskuplja po jediničnoj količini podataka (bajt).

Najudaljenija memorijska struktura će imati najveći kapacitet i najnižu cijenu po jedinici kapaciteta. Cilj ovakve, hijerarhijske, organizacije memorije da se cjelokupna memorija, u odnosu na CPU, ponaša kao da ima približno kapacitet i cijenu najudaljenije, a brzinu najbliže memorije. Ovaj put iluzija pomaže! Razvijene su tehnike koje, u velikoj mjeri, u tome uspjevaju.



Slika 10.1. Prenos podataka između nivoa memorijske hijerarhije

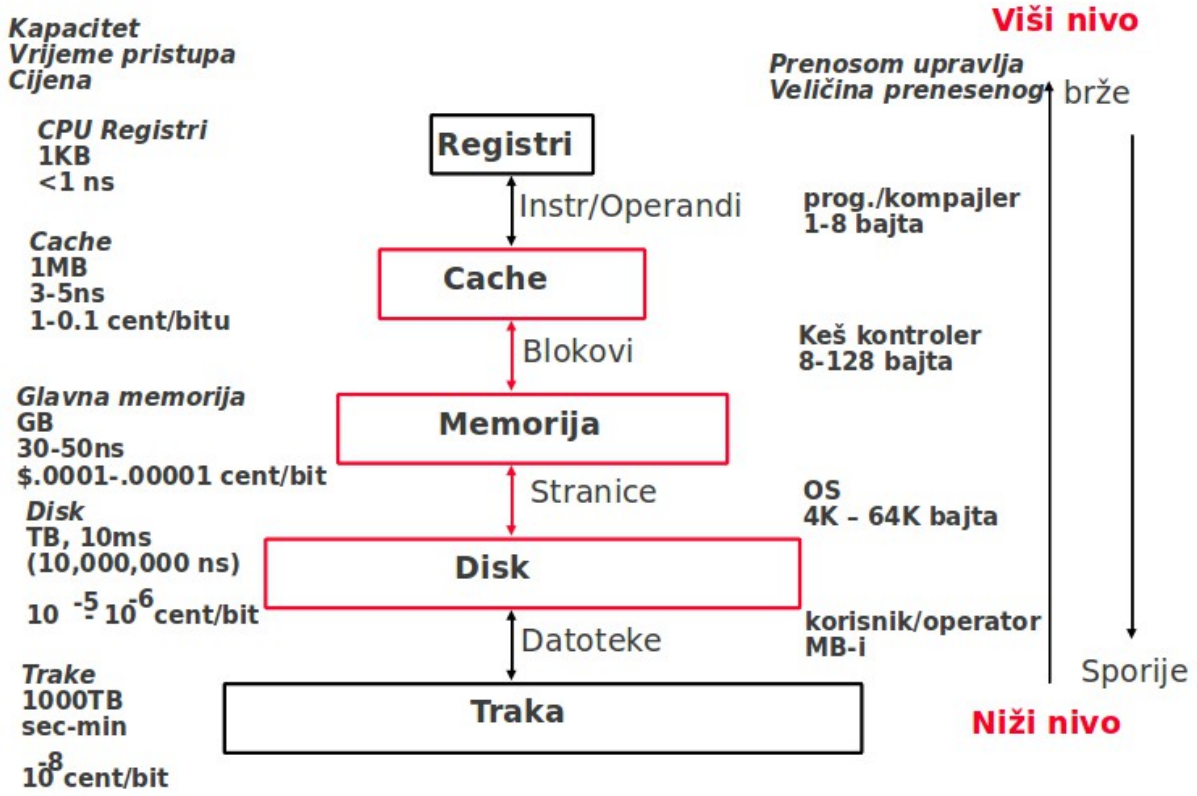
Bez obzira na kom nivou memorijske hijerarhije se traženi podatak nalazi, procesor će najmanje čekati ako ga pročita iz najbrže (najbliže) memorije. Ako ga nađe tamo, to se smatra "pogotkom" (eng. **hit**). U protivnom, to je "promašaj" (eng. **miss**), a blok u kome se nalazi podatak se mora kopirati kroz nivoe hijerarhije prema procesoru do najbrže memorije. Čitavu hijerarhiju treba organizovati tako da se, prosječno, više od 90% slučajeva dešavaju pogodci. Nastoji se da se vjerovatnoća pogotka – nalaženja traženog podatka u najbližoj memoriji, što je više moguće, približi jedinici (100%). U protivnom, podatak se traži u sljedećim nižim (sporijem) nivoima memorijske hijerarhije sve dok se ne pronađe. Zatim se blok u kome se nalazi kopira u sve nivoe do najvišeg. Veličina blokova koji se kopiraju kroz nivoe može varirati. Procesor iz najvišeg nivoa najčešće traži riječ (instrukciju ili podatak). U slučaju promašaja, iz nižih nivoa se dobavljaju blokovi – veće količine podataka, vodeći računa o budućim zahtjevima procesora. Po pravilu, što je niži nivo hijerarhije, kopiraju se veći blokovi instrukcija i podataka.

Keš memorija (od eng. **cache**, skriveno mjesto za čuvanje dragocjenosti) je skupa, brza i (od programera) skrivena memorija najbliža procesoru. Princip keširanja koji je objašnjen, počiva na principu lokalnosti prilikom izvršenja programa. To znači da su pristupi memoriji lokalizovani u vremenu i (memorjskom) prostoru. Vremenska lokalnost predstavlja (visoku) vjerovatnoću da će procesor nakon pristupa jednoj lokaciji u memoriji, u skoroj budućnosti opet njoj pristupiti. Prostorna lokalnost predstavlja (visoku) vjerovatnoću da će procesor nakon pristupa jednoj riječi u memoriji, u skoroj budućnosti pristupiti i njoj susjednim lokacijama u memorijskom prostoru. Primjer za to mogu biti sekvencijalna lokalnost u izvršenju instrukcija (bez grananja) i izvršenje programskih petlji.

Nivo memorijske hijerarhije, M_i	$M_1, M_2, \dots, M_n$
Kapacitet i-tog nivoa, s_i	$s_1 < s_2 < \dots < s_n$
Cijena po bajtu na i-tom nivou, c_i	$c_1 > c_2 > \dots > c_n$
Ukupna cijena, C	$\sum c_i * s_i$
Vrijeme pristupa i-tom nivou, τ_i	$\tau_1 < \tau_2 < \dots < \tau_n$
Vrijeme pristupa do i-og nivoa, t_i	$\tau_1 + \tau_2 + \dots + \tau_n$
Učestanost pogodaka i-tog nivoa, h_i	$h_1 < h_2 < \dots < h_n = 1$
Učestanost promašaja do i-tog nivoa, m_i	$(1-h_1)(1-h_2) \dots (1-h_{i-1})$
Efektivno vrijeme pristupa, T	$\sum m_i * \tau_i$

Tabela 10.1. Pojednostavljen prikaz elemenata memorijske hijerarhije

Glavni cilj projektanta memorijske hijerarhije je da postigne što kraće efektivno vrijeme pristupa T, za što nižu ukupnu cijenu C. Ovo je, naravno, mnogo lakše reći nego uraditi, jer je čitava analiza prilično pojednostavljena – za potrebe ovoga udžbenika.
U realnosti, ovi hipotetski nivoi memorijske hijerarhije se preslikavaju na keš memorije (prvih nekoliko nivoa najbližih procesoru), radnu memoriju i virtuelnu memoriju. Nešto detaljniji prikaz nivoa memorijske hijerarhije sa kratkim opisom komunikacije među njima je dan na slici 10.2.

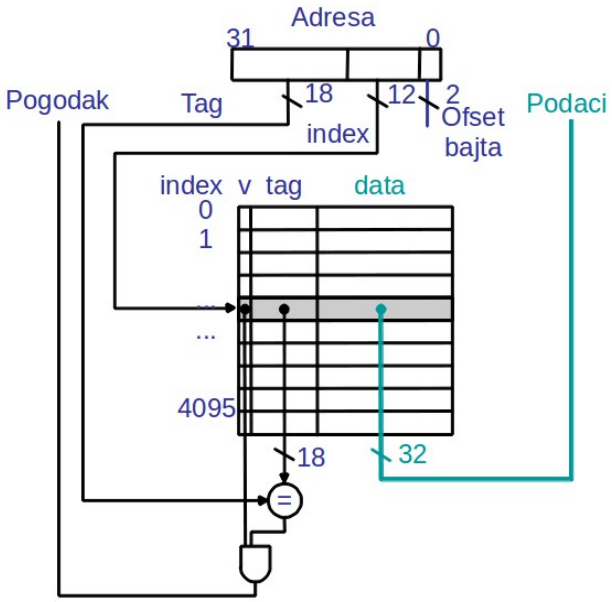


Slika 10.2. Detaljniji opis nivoa u memorijskoj hijerarhiji, sa cijenama, kapacitetima, inicijatorima prenosa i veličinama prenesenih blokova

Radi jednostavnosti prikaza osnovnih pojmova keširanja, neka je keš jednonivoski i nepodijeljen – kešira i instrukcije i podatke. Kako je kapacitet keša za dva do tri reda veličine manji od kapaciteta glavne memorije, jasno je da može sadržati samo mali dio podataka kopiranih iz nje. Zbog toga se nameću tri osnovna pitanje pri projektovanju keša:

1. koji dio podataka iz glavne memorije kopirati u keš,
2. koliki dio tih podataka kopirati i
3. gdje u keš kopirati odabrane podatke?

Najjednostavniji način ovog preslikavanja se zove **direktno preslikani keš**. On podrazumjeva da svaka riječ (ili blok riječi) iz glavne memorije može biti kopiran – preslikan samo na jedno mjesto u kešu.



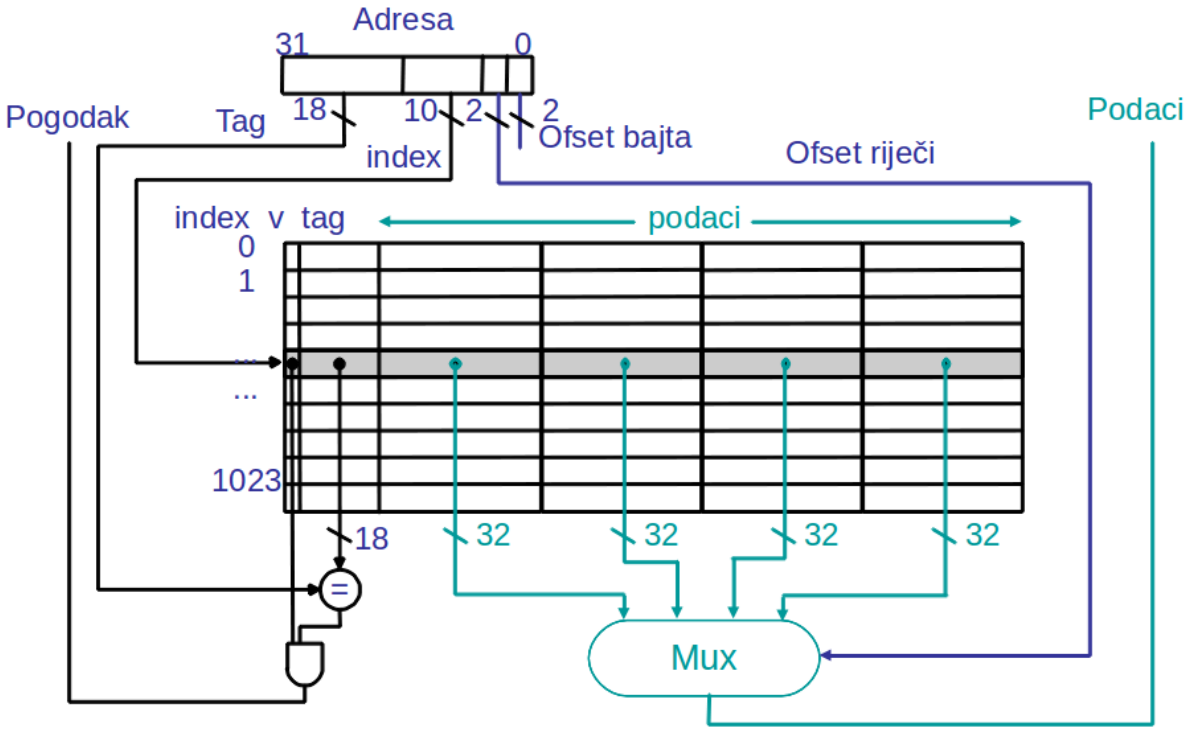
Slika 10.3. Direkno preslikani keš. 4GB glavne memorije se preslikava u 4K 32-bitnih riječi.

Svaka 32-bitna adresa bajta iz 32-bitne glavne memorije, prikazana u gornjem dijelu slike 10.3. se, u ovom primjeru, dijeli na:

- 2 bita polja bita ofseta, kako bi procesor mogao izdvojiti traženi bajt iz 4-bajtne riječi -bloka,
- 12 bita polja indexa, kojim se jednoznačno određuje jedino od 4K mjesta u ovakvom kešu na koje se može preslikati traženi blok iz memorije i na kome se može kasnije naći i
- 18 bita polja tag-a, koje se upisuje u polje tag izabrane linije u kešu (kod preslikavanja bloka iz glavne memorije) ili se poredi sa ranije upisanom vrijednošću na istom mjestu (kod pretraživanja keša). U svakoj liniji ovakvog keša se mogu naći preslikani podaci sa po 2^{18} = 256K različitih mjesta (adresa) iz glavne memorije i međusobno će se razlikovati po sadržaju tag-polja svoje adrese.

Svaka linija u kešu se sastoji od 32-bitne keširane riječi (instrukcije ili podatka), ali i od 18-bitnog tag-a i jednog bita ispravnosti keširane riječi – validnosti V.

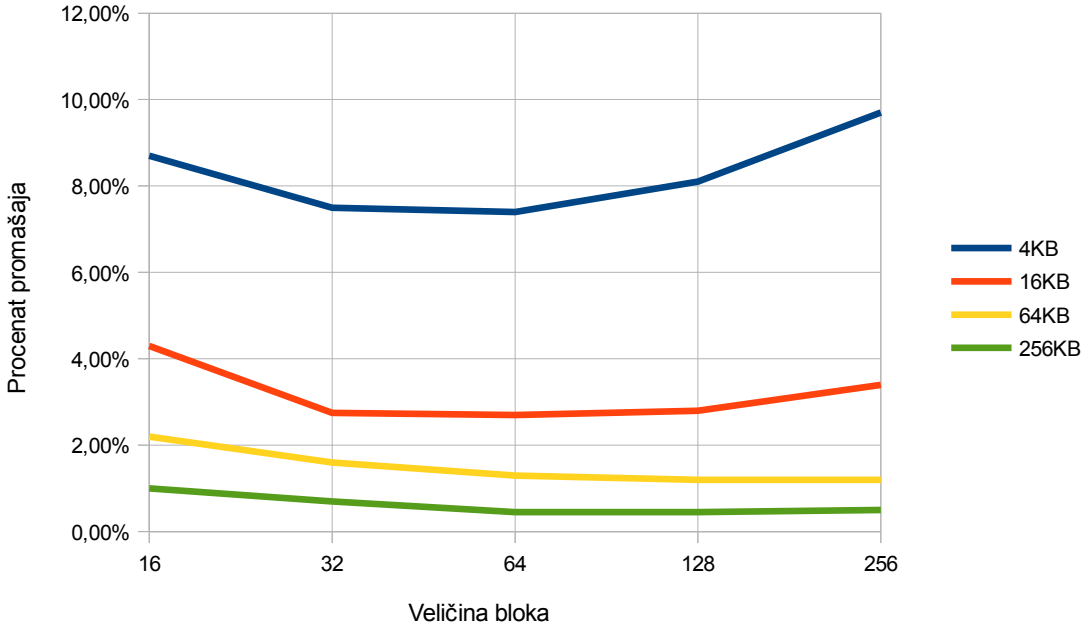
Na početku rada računara, keš je prazan i svi V biti u njemu su u neaktivnom stanju. Kako se keš puni, biti validnosti se postavljaju. Sve lokacije u kojima je V bit neaktivan, smatraju se praznim lokacijama, a ostale sadrže korisne kopije instrukcija ili podataka iz glavne memorije.
Ako je blok, kao jedinica za prenos između nivoa u memorijskoj hijerarhiji, veći od jedne riječi, direktno preslikani keš ima strukturu kao na slici 10.4.



Slika 10.4. Direktno preslikani keš sa blokom od 4 riječi

Svaki put kada se desi promašaj u kešu, u njega se ne donosi samo tražena riječ, već čitav blok iz glavne memorije u kome se ta riječ nalazi, u nadi da će i ostale riječi uskoro zatrebati (zbog prostorne lokalnosti u izvršavanju programa). Struktura i kapacitet keša su isti kao na slici 10.3. samo je on proširen horizontalno. Uz isti kapacitet keša, broj linija je sada četiri puta manji (jer je kapacitet bloka četiri riječi – četiri puta veći). Time je polje bita index smanjeno za dva bita, koji postaju ofset riječi unutar bloka. Bilo bi pogrešno zaključiti da se povećanjem bloka smanjuje procenat promašaja u kešu, iako veći blok sadrži više lokalnih instrukcija i/ili podataka iz programa koji se izvršava. Povećanje bloka smanjuje broj blokova koji se mogu naći u kešu, pa se asocijativnost smanjuje. Slika 10.5. prikazuje promjene procenata promašaja u kešu u zavisnosti od veličine bloka. Četiri boje pretstavljaju trendove za različite kapacitete keš memorija. Može se vidjeti da procenti promašaja rastu kada blokovi postanu preveliki u odnosu na kapacitet keša. To se posebno dobro vidi za keševe malog kapaciteta. Naprimjer, u keš kapaciteta 4KB mogu stati samo četiri bloka od po 256 četvorobajtnih riječi. Prema tome, kopiran je mali broj lokacija – asocijativnost keša je mala. Što je blok veći, promašaj u kešu će zahtjevati više vremena dok se novi (veliki) blok preslika u keš, što će direktno uticati na performanse procesora (njegov CPI). S druge strane, sa povećanjem kapaciteta keša, dobavlja se sve više i više informacija, pa procenat promašaja pada. Dijagram na slici 10.5. generalizuje pomenute trendove i ne uzima u obzir

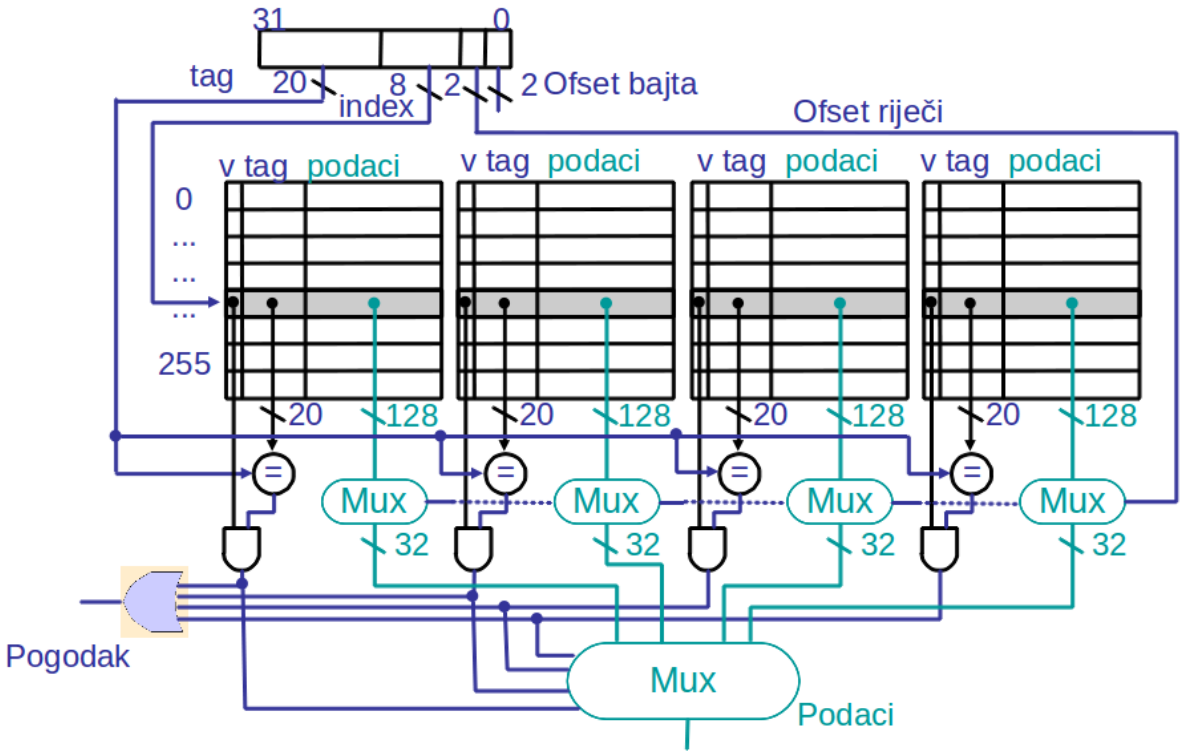
asocijativnost keševa.



Slika 10.5. Procenat promašaja u kešu u zavisnosti od veličine bloka (u riječima) i kapaciteta keša (u KB)

Problem sa direktno preslikanim kešom je taj što se, nakon promašaja, novi blok iz memorije može smjestiti samo na jedno mjesto. Na tom mjestu se može nalaziti najsvježiji blok u čitavom kešu, pa, po principu vremenske lokalnosti, gubi se (izbacuje) blok koji ima velike šanse da uskoro bude potreban.

Povećanje asocijativnosti direktno preslikanog keša se postiže tako što se dva, četiri, ili više takvih keševa koristi paralelno. Time se izbor mjesta za smještaj novog bloka povećava. Na slici 10.6. je prikazana struktura 4-struko **grupno asocijativnog keša**, istog kapaciteta i veličine bloka kao u prethodnom primjeru. On radi tako da se svaki blok iz glavne memorije može naći na četiri mjesta u kešu. To pokreće dva nova pitanja – gdje (u koje od 4 mjesta) u kešu preslikati novi blok iz memorije, i kako naći traženi podatak koji se sada može naći u liniji koja sadrži četiri bloka.



Slika 10.6. Četvorostruko grupno asocijativni keš

Kao što je prikazano na slici 10.6., gornjih 30 bita adrese, kojima se adresira riječ u memoriji, se sada dijele na dva bita ofseta riječi, koji određuju položaj tražene riječi u bloku od četiri riječi, 8 bita index-a za izbor jedne od 256 lijia u kešu i preostalih 20 bita tag-a.

Postoji više načina određivanja mjesta za preslikavanje novog bloka. Ukoliko postoji prazno mjesto u izabranoj liniji keša, ono je logičan izbor za upis novog bloka. Ukoliko su svi blokovi u liniji zauzeti, tada se mjesto bira na jedan od načina:

1. slučajnim izborom (eng. Random)
2. zamjenjuje se najstariji blok (eng. FIFO, od First In First Out) ili
3. zamjenjuje se blok koji je najmanje korišten u posljenje vrijeme (LRU, od eng. Least Recently Used).

U druga dva pomenuta načina, potrebno je obezbjediti mehanizme egzaktnog određivanja apsolutne ili relativne starosti bloka (pomoću niza brojača i komparatora proteklog vremena itd.).

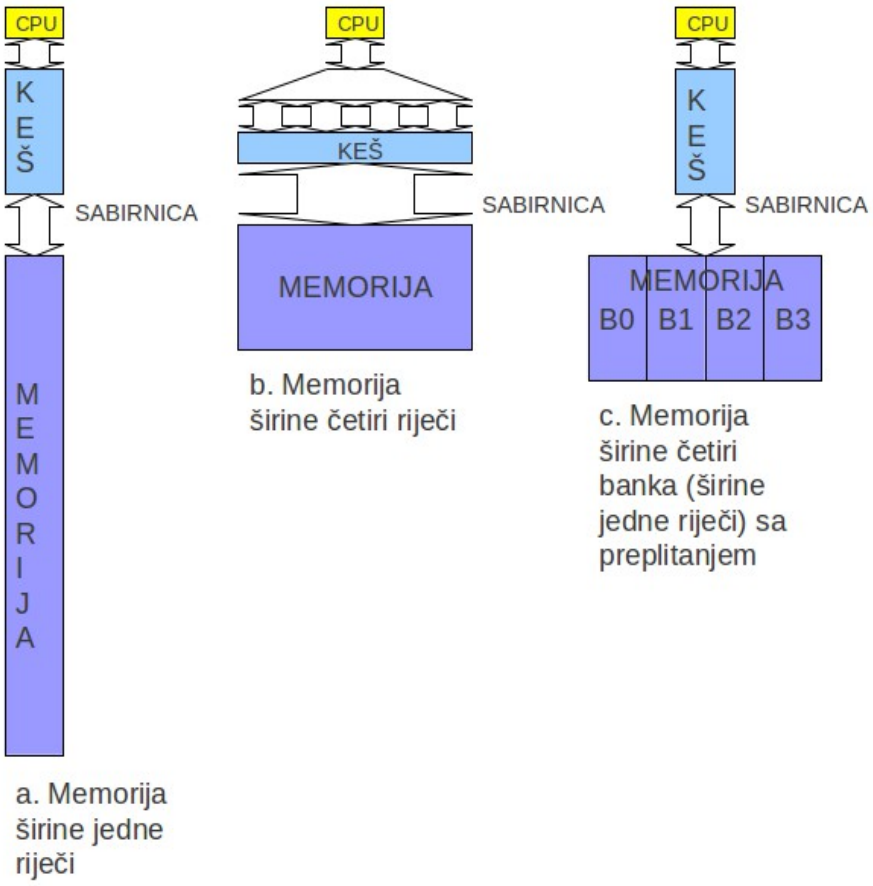
Odgovor na drugo postavljeno pitanje je da se pretraživanje ova četiri (pod)keša koristi četiri komparatora polja bita tag-a, koji rade paralelno, kao i složeniji mehanizam za signalizaciju pogotka i proslijeđivanje nađene riječi procesoru.

Dalje udvostručavanje asocijativnosti (broja paralanih keševa), za isti kapacitet keša, smanjivao bi broj bita index-a, a povećavao broj bita taga.

Potpuno asocijativni keš bi, u tom slučaju, imao samo jednu liniju sa 1024 bloka čije identitete bi osiguravali 28-bitni tag-ovi. Tada bi se svaki novi blok iz memorije mogao upisati u bilo koji od 1K blokova u kešu, ali bi bilo neophodno koristiti isto toliko komparatora za poređenje tagova, kao i vrlo složen mehanizam za signalizaciju pogotka i proslijeđivanje nađene riječi procesoru.

U praksi se vrlo rijetko koristi direktno preslikani keš (zbog slabe asocijativnosti), kao i potpuno asocijativni keš (zbog svoje složenosti). Obično se prvi nivo keša, najmanjeg

kapaciteta i najbliži procesoru, realizuju kao 4-struko grupno asocijativni (kompromis asociativnosti i jednostavnosti-brzine), dok se keševi drugog i (eventualno) trećeg nivoa realizuju kao višestruko grupno asocijativni (asocijativniji i sporiji). Ovakav pristup ima smisla ako se zna da je zadatak najbližeg nivoa keša da ono što ima isporuči procesoru brzo – da smanji kašnjenje, dok je zadatak ostalih nivoa da smanje procenat promašaja – usluže što veći procenat promašaja u prvom nivou keša.



Slika 10.7. Različiti načini povezivanja keša i glavne memorije

U trenutku nastanka promašaja u kešu, u njega se donosi novi blok od nekoliko riječi, među kojima i ona koju procesor upravo traži. Donošenje bloka se može izvršiti na više načina (Slika 10.7.). Prvi pristup (a) je da su sve sabirnice iste širine i da se blok prenosi sukcesivnim prenosom riječ-po-riječ od glavne memorije prema kešu i obratno. Drugi pristup (b) zadržava sabirnicu od keša ka procesoru širine jedne riječi, ali za prenose između keša i glavne memorije koristi proširenu sabirnicu, kojom se blok prenosi u jesnom memorijskom ciklusu. U tom slučaju je i memorijska riječ proširena na širinu bloka (u ovom slučaju 4 riječi). Treći pristup (na slici pod c) je da se zadrže sabirnice širine jedne riječi (kao pod a) ali se preplitanjem memorijskih modula (bankova) "sakrije" kašnjenje druge, treće i četvrte riječi iz bloka prilikom prenosa. Kako je vrijeme pristupa memoriji znatno veće od vremena prenosa jedne riječi preko sabirnice, treći pristup omogućava vrijeme prenosa bloka slično kao pod (b) uz zadržavanje jednostavnog ožičenja (uskih sabirnica) kao pod (a).

Primjer:
Neka je potreban 1 ciklus za slanje adrese, 15 ciklusa za pristup DRAM-u i 1 ciklus za prenos

jedne riječi podatka.
Neka se blok keša sastoji od 4 riječi, a DRAM je širine jedne riječi, tada je kašnjenje kod promašaja u kešu sa pristupom (kao pod a):

$$K_a = 1 + 4x15 + 4x1 = 65 \text{ ciklusa} .$$

Neka je širina memorije i sabirnice 4 riječi – (kao pod b), tada je kašnjenje kod promašaja:

$$K_b = 1 + 15 + 1 = 17 \text{ ciklusa}.$$

Korištenjem četiri memorijska modula/banka i sabirnice širine jedne riječi (kao pod c), kešnjenje kod promašaja u kešu će biti:

$$K_c = 1 + 1x15 + 4x1 = 20 \text{ ciklusa}.$$

Kao što se može vidjeti, ovakav kompromis (jednostavniji hardver sa nešto većim kašnjenjem) se isplati, pa većina današnjih računara koristi preplitanje bankova u memoriji – treći pristup.

10.2. Performanse keš memorije

U tabeli 9.1. je data kratka analiza performansi n-nivoske memorijske hijerarhije. Prikazane su cijene pojedinih nivoa kao i njihovi kapaciteti i kašnjenja koja unose.
Radi jednostavnosti, neka se memorijska hijerarhija sastoji od samo dva nivoa – jednog keša (M_1) i glavne memorije (M_2). Kako je vrijeme pristupa kešu

$$t_1 = \tau_1$$

a vrijeme prostupa memoriji

$$t_2 = \tau_1 + \tau_2$$

uz vjerovatnoću pogodaka po nivoima

$$h_1 < h_2 = 1$$

jer se pretpostavlja da je M_2 zadnji nivo u hijerarhiji, a vjerovatnoća promašaja do nivoa 1

$$m_1 = 1$$

jer nema nultog nivoa, a do nivoa 2

$$m_2 = 1 - h_1$$

efektivno vrijeme pristupa ovakvoj memorijskoj hijerarhiji će biti

$$T = m_1 h_1 t_1 + m_2 h_2 t_2 =$$
$$= h_1 t_1 + (1 - h_1) t_2 =$$

$$= \tau_1 + (1-h_1)\ \tau_2$$

To znači da će se, u svakom pristupu memorijskoj hijerarhiji, svakako potrošiti vrijeme pristupa kešu (pogotka) τ_1 a samo u slučaju promašaja u kešu (čija je vjerovatnoća $1-h_1$) i vrijeme kašnjenja zbog promašaja τ_2 .
Kao što je već pomenuto u jednom od ranijih poglavlja, brzina kojom procesor može da izvrši neki program zavisi od broja instrukcija u programu (N), broja ciklusa sata potrebnih za izvršenje jedne instrukcije (CPI) i frekvencije sata (f) na kojoj radi procesor. Vrijeme izvršenja programa (T_p) se može izračunati kao

$$T_p = \frac{\text{Instrukcija}}{\text{Programu}} \times \frac{\text{Ciklusa sata}}{\text{Instrukciji}} \times \frac{\text{Sekundi}}{\text{Ciklusu sata}}$$

ili kao

$$T_p = N*CPI/f$$

Sa stanovišta memorijske hijerarhije, na N i f se ne može uticati. Na CPI može i to tako da se realni CPI razloži na idealni CPI koji zavisi od organizacije protočne strukture i učestanost zastoja zbog pristupa memoriji po instrukciji. Drugim riječima

$$\frac{\text{Mem. zastoji}}{\text{Instrukciji}} = \text{Učestanost promašaja} \times \text{Kašnjenje kod promašaja} \times \frac{\text{Br. pristupa mem.}}{\text{Instrukciji}}$$

Prema tome, za poboljšanje performansi memorijske hijerarhije (a time i para procesor-memorija) potrebno je smanjiti učestanost promašaja, kašnjenje kod promašaja i, po mogućnosti, kašnjenje kod pogotka u kešu.
Kašnjenje kod promašaja zavisi od toga kako se prenose podaci između glavne memorije i keša. Slika 9.7. prikazuje tri osnovna načina takvog prenosa. Sa druge strane, savremene komponente DRAM-a i moduli podržavaju (pored preplitanja) strananje – brži pristup elementima na već otvorenoj stranici. Osnovni memorijski elementi (ćelije) su organizovane u 2-dimenzionalne strukture – matrice. Kod pristupa bilo kojem elementu, adresira se prvo red - čitav red matrice u kojem se nalazi traženi element se interno kopira u bafer. Nakon toga se adresira kolona – element unutar reda. Pristup drugim elementima iz dobavljenogg (baferovanog) reda (stranice) je brži, jer se ne mora ponovo adresirati red - stranica. Najefikasniji su sekvencijalni pristupi u okviru otvorene stranice – potrebna je samo početna adresa bloka, ostale se podrazumjevaju. Slučajni pristupi elementima unutar otvorene stranice (reda) zahtjevaju samo mijenjanje adrese kolone (elmenta u redu).
Ovim je opisan osnovni način funkcionisanja keš podsistema unutar memorijske hijerarhije. Unutar njega postoji još mnogo različitih načina čitanje, pisanja i zamjene blokova u kešu, koje mogu značajno uticati na performanse računarskog sistema.
Čitanje iz memorijske hijerarhije se može vršiti sekvencijalno ili paralelno, a kod promašaja u kešu sa ili bez ubrzanog prosljeđivanje tražene riječi procesoru. **Sekvencijalno** čitanje podrazumjeva pokretanje pristupa memoriji tek kada se desi promašaj u kešu. **Paralelno** (konkurentno) čitanje, pored pristupa kešu istovremeno pokreće (preventivno) čitanje i iz memorije. U slučaju pogotka u kešu, otkazuje se pristup memoriji. Kašnjenje kod pogotka je isto u oba slučaja, ali se konkurentnim čitanjem skraćuje kašnjenje podataka nakon promašaja u kešu. Nedostatak konkurentnog čitanja je veće (češće) zauzeće sabirnice prema memoriji.

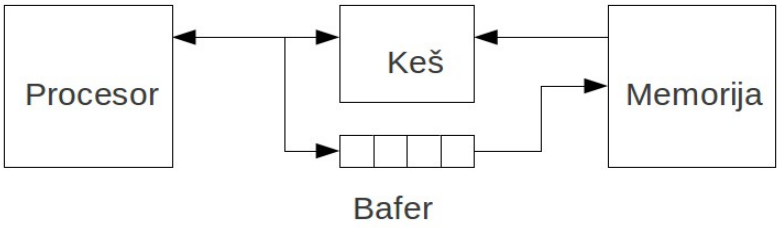
Kod promašaja u kešu, novi blok iz memorije koji sadrži traženu riječ se može prvo upisati u keš, a zatim traženu riječ poslati procesoru – **bez ubrzanog prosljeđivanja**. Drugi način je da se, tokom čitanja bloka iz memorije, tražena riječ istovremeno šalje procesoru i upisuje u keš. Najbrži način je da se iz memorije prvo čita tražena riječ, koja se istovremeno prosljeđuje procesoru i upisuje u keš – **sa ubrzanim prosljeđivanjem**, a zatim se prenose ostale riječi traženog bloka iz memorije u keš.

Dobavljanje iz memorije se može pokretati na više načina. Osnovni je da se novi blok donosi u keš tek **nakon zahtjeva procesora** i promašaja u kešu. Drugi način je da se, pored bloka u kome se nalazi tražena riječ, izvrši **preddobavljanje** (eng. prefetching) i sljedećeg ili nekoliko sljedećih blokova, u nadi da će se sekvencijalno pristupati riječima u tim blokovima. Preddobavljanje se može pokrenuti **hardverski** – kontrolerom keša, ili **softverski**, kada programer ili kompajler odlučuju o tome. Softversko preddobavljanje se obavlja tako da se u programu rasporede instrukcije koje će izazvati promašaj u kešu, kako bi se unaprijed dobavljali blokovi koji će kasnije biti korišteni iz keša. Ova napredna tehnika unosi dva nova problema - koliko blokova dobavljati unaprijed (koliko daleko ići) i kada (koliko često) vršiti preddobavljanje. Oboje zavisi od strukture programa i podataka koje on obrađuje, pa loša procjena može dovesti do gubljenja vremena i prostora u kešu, kao i nepotrebne zauzetosti sabirnice prema memoriji.

Zamjena bloka u kešu se dešava svaki put kada se desi promašaj i novi blok se dobavi iz memorije. Njega treba smjestiti, a neki drugi vratiti u memoriju (ako je modifikovan) ili odbaciti. Kod direktno preslikanog keša novi blok ima samo jedno mjesto u kešu. Međutim, kod keša sa većim stepenom asocijativnosti, javlja se problem na koje od dva ili više mjesta smjestiti novi blok, odnosno, koji blok njime zamijeniti. Ovaj problem se može riješiti na više načina. Najčešće korišteni način je da se novi blok stavi na mjesto najmanje korištenog bloka u zadnje vrijeme (eng. Least Recently Used – **LRU**). Da bi to bilo moguće, pored svakog bloka treba imati brojač proteklog vremena od zadnjeg pristupa, kao i komparator vrijednosti pored svih brojača koji su kandidati za zamjenu. Blok čiji brojač je najdalje odmakao sa brojanjem je najmanje korišten u posljednje vrijeme, pa komparator nema problema da ga locira kao blok koji treba zamijeniti novim. Drugi način je da se vrši zamjena bloka koji je najrjeđe korišten u zadnje vrijeme (eng. Least Frequently Used – **LFU**). Blok čiji brojač pristupa sadrži najmanju vrijenost će, u ovom slučaju, biti zamijenjen novim. Treći način je da se zamjenjuje blok koji je proveo najviše vremena u kešu – najstariji blok (eng. First in First Out – **FIFO**). Tehnički najmanje zahtjevan način određivanja bloka za zamjenu je slučajnim odabirom (eng. **Random**). Osim jednostavnosti (brzine) ovaj način nema drugih dobrih osobina.

Pisanje u memorijsku hijerarhiju može rezultirati pogotkom ili promašajem u kešu. U slučaju pogotka (eng. **Write Hit**), kada se blok u koji se piše nalazi u kešu, upis se može obaviti ažururanjem bloka u kešu i riječi u memoriji. Ovaj način pisanja se zove pisanje kroz keš (eng. **Write Through**). Prema ranije pomenutim zakonima prostorne i vremenske lokalnosti, velike su šanse da će procesor u skoroj budućnosti ponovo čitati ili pisati vrijednost koju upisuje. Zato bi bilo logično odgoditi pisanje u memoriju dok god je to moguće. Tek kada je modifikovani blok određen za izbacivanje iz keša (radi oslobađanja prostora za novi blok), vrši se njegovo kopiranje u memoriju. Takav način pisanja se zove pisanje nazad (eng. **Write Back**) kada procesor nastavlja čitati i pisati keširani podatak i odgađanjem upisa u memoriju smanjuje saobraćaj na sabirnici. U slučaju promašaja u kešu (eng. **Write Miss**) pisanje se može završiti na dva osnovna načina. Prvi je da se blok iz memorije, u koji se želi pisati, prvo dobavi u keš, a zatim se upis nastavi kao kod pogotka u kešu (eng. **Write Alocate**). Drugi način je da se upis obavi samo u memoriji (eng. **Write Update**), a da se blok kešira samo

kada mu se pristupi radi čitanja. Na taj način se štedi vrijeme keširanja bloka, ali se izlaže riziku skorog promašaja kod čitanja upisanog podatka. U oba slučaja pisanja u memoriju (pisanja kroz keš i pisanja nazad) procesor koristi bafer za sakrivanje kašnjenja (slika 10.8.).

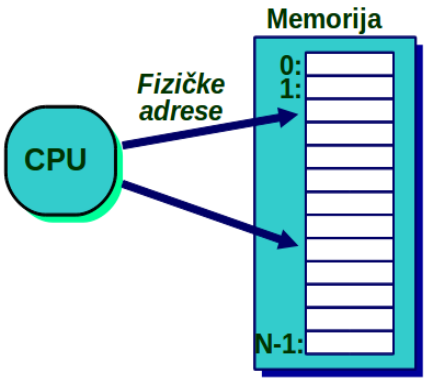


Slika 10.8. Delegiranje pisanja u memoriju - sakrivanje kašnjenja kod pristupa memoriji

Pisanje u bafer je brz i, delegiranjem jednog ili više upisa u memoriju, procesor se rasterećuje čekanja – kašnjenja zbog pristupa memoriji. Kod pisanja kroz keš, procesor upisuje riječ, dok kod pisanja nazad keš kontroler piše čitav blok iz keša. U slučaju da procesor uskoro ponovo traži ono što je upisano u bafer, mora postojati mehanizam pretraživanja bafera u slučaju promašaja u kešu. U slučaju pronalaženja traženog podatka u baferu, on se može ili direktno proslijediti procesoru i ponovo keširati ili zaustaviti procesor dok se bafer ne isprazni, pa ponoviti keširanje iz memorije.
Po opštoj organizaciji, keš podsistemi se mogu podijeliti na više vrsta. Po namjeni se keševi mogu podijeliti na keševe **instrukcija** i keševe **podataka**. Po pravilu, keširanje instrukcija je jednostavnije jer se instrukcije samo čitaju – jednosmjerno keširanje. Keširanje podataka je dvosmjerno i mora uključivati mehanizam označavanja modifikovanih blokova (eng. **dirty bit**). Keširanje instrukcija i podataka se može vršiti u odvojenim (eng. **split**) ili objedinjnim (eng. **unified**) keševima. Prvi nivo keševa je u savremenim procesorima razdvojen radi paralelizma u radu i otklanjanja strukturnih hazards u protočnim strukturama. U tom slučaju su oba keša optimizirani za svoje uloge. Nedostatak ovog pristupa je mogućnost slabog iskorištenja jednog od keševa a preopterećenje drugog (veliki programi sa malom količinom podataka ili obratno). Preostali nivo(i) keševa se najčešće realizuju kao jedinstveni – dijeljeni u keširanju i instrukcija i podataka, čime se bolje iskorištava njihov ukupni kapacitet. Istorija razvoja mikroprocesora pokazuje da su prvi procesori sa kešom (instrukcija) imali **jednonivoski** keš (brz i malog kapaciteta) integrisan na istom silicijumu (eng. **on-chip**). Kasnije, sa povećanjem razlike u brzini rada procesora i memorije, pojavili su se **višenivoski** keševi čiji nivoi, osim prvog, sporiji i većeg kapaciteta, su u početku bili realizovani van procesora (eng. **off-chip**), a kasnije integrisani na istoj podlozi.
Bez obzira na tip keša, njegovu organizaciju, asocijativnost i kapacitet, postoje tri osnovna razloga zbog kojih su promašaji u kešu neizbježni. Prvi razlog su promašaji koji nastaju prilikom inicijalizacije – početka rada računarskog sistema, kada je keš prazan (eng. **Cold start**). Kako se keš puni, ovaj tip promašaja se smanjuje. Drugi razlog su promašaji koji nastaju zbog značajne razlike u kapacitetu keša u odnosu na memoriju (eng. **Capacity miss**). Keš ne može sadržati sve što se nalazi u memoriji. Treći razlog neizbježnog nastanka promašaja u kešu su konflikti među većim brojem blokova iz memorije koji se “bore” za mjesto u kešu (eng. **Conflict miss**). Ovi konflikti nastaju kao posljedica određenog načina preslikavanja blokova iz memorije u keš – što je viši nivo asocijativnosti, to su ovi konflikti manji. Kod potpuno osocijativnog keša poslije inicijalnih promašaja, nema promašaja zbog konflikata, veš samo zbog (ograničenog) kapaciteta.

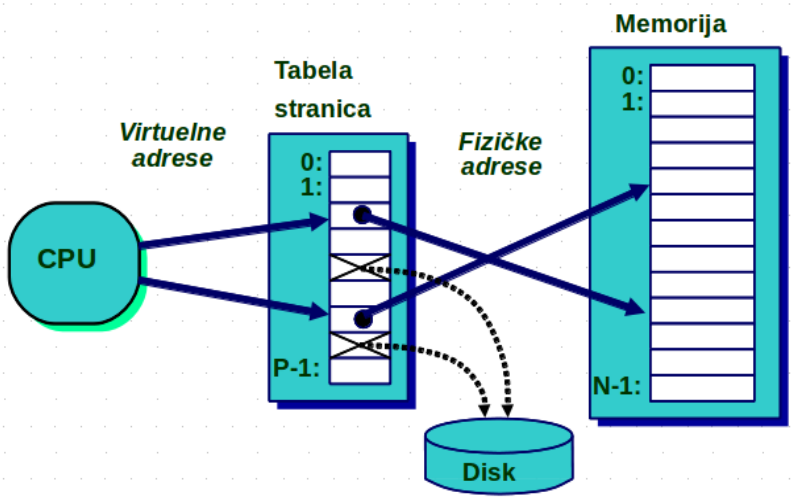
10.3. Virtuelna memorija

Kao što keš omogućava brzi pristup dijelovima programa i podataka iz memorije koji se najčešće koriste, tako i memorija može "keširati" dijelove diska koji se često koriste. Ova tehnika se naziva virtuelna memorija i ima zadatak da programeru (i njegovom programu) da iluziju da mu je na raspolaganju proširena (praktično neograničena) radna memorija, zaštićena od pristupa drugih korisničkih programa.



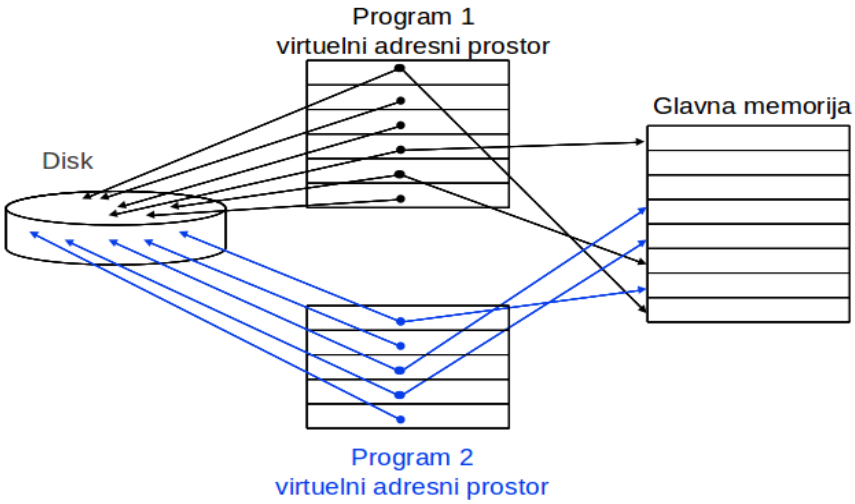
Slika 10.9. Procesor generiše adrese koje se direktno odnose na riječi ili bajte u fizičkoj memoriji. Ovako su radili personalni računari do polovine 1980-tih godina, većina Cray vektorskih superračunara i većina jednostavnih ugrađenih računara.

Potrebno je da **fizička memorija** sadrži samo aktivne dijelove različitih programa, kako bi se efikasno dijelila kao dragocjen resurs, slično kešu. Na taj način više programa efikasno dijeli i procesor i memoriju.
Kako se, u trenutku kompajliranja programa, ne zna u kom dijelu fizičke memorije će se on izvršavati i s kojim drugim programima će dinamički dijeliti memoriju, potrebno je program i kompajler osloboditi ovih briga. Kompajler ne mora da generiše **relokatibilan kod** – kod koji ne zavisi od mjesta u memoriji na kome se porkreće (koristi relativne skokove i grananja), niti da vodi računa o količini raspoložive fizičke memorije. Virtuelna memorija obezbjeđuje svakom programu logički odvojen i njegov vlastiti **virtuelni adresni prostor** kome drugi programi ne mogu pristupiti. To se postiže prevođenjem programskog adresnog prostora u **fizički adresni prostor**. Ovim mehanizmom se može programima dati iluzija da raspolažu adresnim prostorom većim od raspoloživog fizičkog adresnog prostora pa programeri ne moraju eksplicitno rješavati problem izvršenja programa većeg od raspoložive memorije, njegovog particioniranja i upravljanja prebacivanjem iz memorije na disk i obratno (eng. **overlays**). Kao i kod keševa, i virtuelni adresni prostor i fizički adresni prostor su podjeljeni na blokove. Blokovi se u ovom kontekstu nazivaju stranicama, a promašaj u memoriji se zove **greška stranice** (eng. page fault). Procesor emituje **virtuelnu adresu**, koja se hardversko-softverski prevodi u **fizičku adresu**, koja se koristi za prostup fizičkoj memoriji. Slika 10.10. prikazuje sistem sa virtuelnom memorijom.



Slika 10.10. Sistem sa virtuelnom memorijom. Ovako rade savremeni personalni računari, radne stanice i serveri.

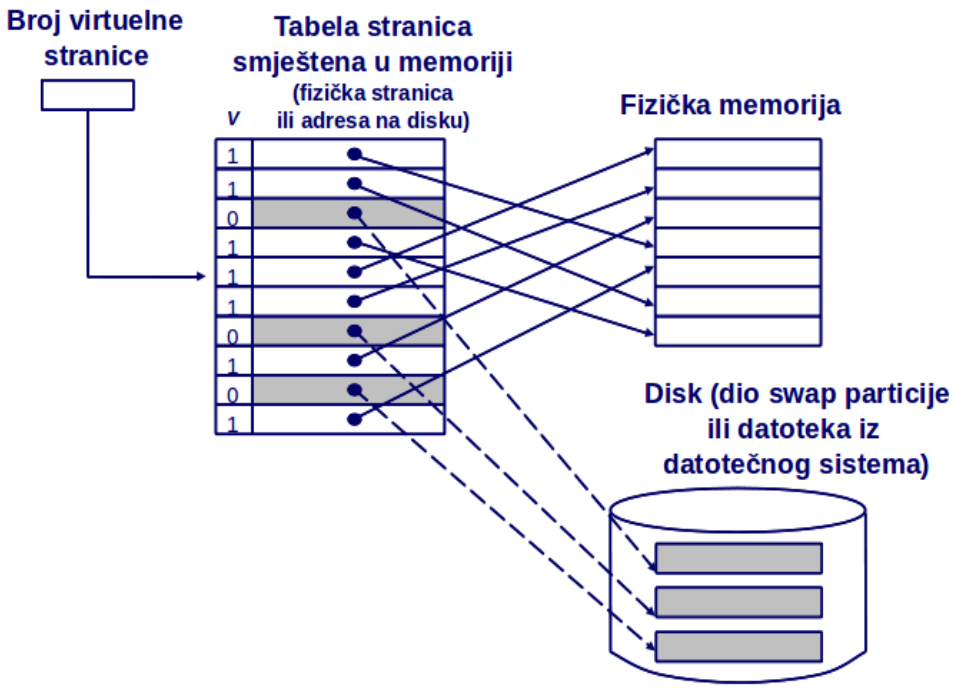
Virtuelna memorija se najčešće oslanja na diskove, kao sljedeći niži nivo u memorijskoj hijerarhiji, jer zadržavaju svoj sadržaj i nakon nestanka napajanja i, po pravilu, imaju mnogo veći kapacitet od fizičke radne memorije (DRAM-a). Može se pretpostaviti da su, inicijalno, sve stranice virtuelne memorije smještene na disku. Samo njihov aktivni podskup je preslikan i u glavnu memoriju i dinamički se mijenjaju (po potrebi). Da bi ovo bilo moguće, arhitektura skupa instrukcija (ISA) mora podržavati veliki adresni prostor, a prevođenje adresa mora podržati specijalni hardver (tabele, komparatori itd.) kao i softver (operativni sistem).



Slika 10.11. Dva programa dijele isti adresni prostor u glavnoj memoriji. Samo aktivni dijelovi programa se drže u glavnoj memoriji, a ostali na disku.

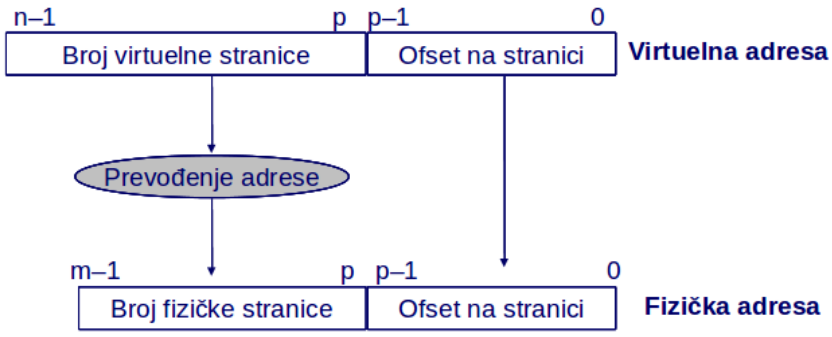
Na slici 10.11. je prikazano kako dva programa dijele isti fizički adresni prostor istovremeno. Programski (virtuelni) adresni prostor se može dijeliti na stranice (nepromjenjive dužine) ili segmente (promjenjive). Iako segmentirana memorija ima neke prednosti u odnosu na onu sa stranicama (efikasnije korištenje fizičke memorije), upravljanje segmentiranom memorijom

je složeniji zadatak za operativni sistem (defragmentacija i optimizacija korištenja memorije), u daljem tekstu će biti govora o virtuelnoj memoriji sa stranicama.
Aktivnih programa može biti i više, a svaki ima svoju tabelu stranica u kojoj se nalaze, za svaku virtuelnu stranicu, pokazivači na odgovarajuću fizičku – u fizičkoj memoriji ili na disku (slika 10.12.). Tabela stranica je indeksirana brojem virtuelne stranice i pokazuje na redni broj fizičke stranice ako je ona u memoriji (V bit postavljen) ili adresu na disku gdje se stranica nalazi.



Slika 10.12. Tabela stranica pokazuje gdje se nalazi odgovarajuća fizička stranica u memoriji (V bit postavljen) ili adresu na disku.

Prevođenje virtuelnih u fizičke adrese se svodi na pretraživanje tabele stranica kako bi se u virtuelnoj adresi broj virtuelne stranice zamijenio brojem odgovarajuće fizičke stranice, ako se ona nalazi u memoriji (ako je bit validnosti V postavljen). Ostatak virtuelne adrese (na slici 10.13. donjih *p* bita) se ne mijenja, jer oni predstavljaju ofset traženog bajta ili riječi unutar virtuelne i fizičke stranice.

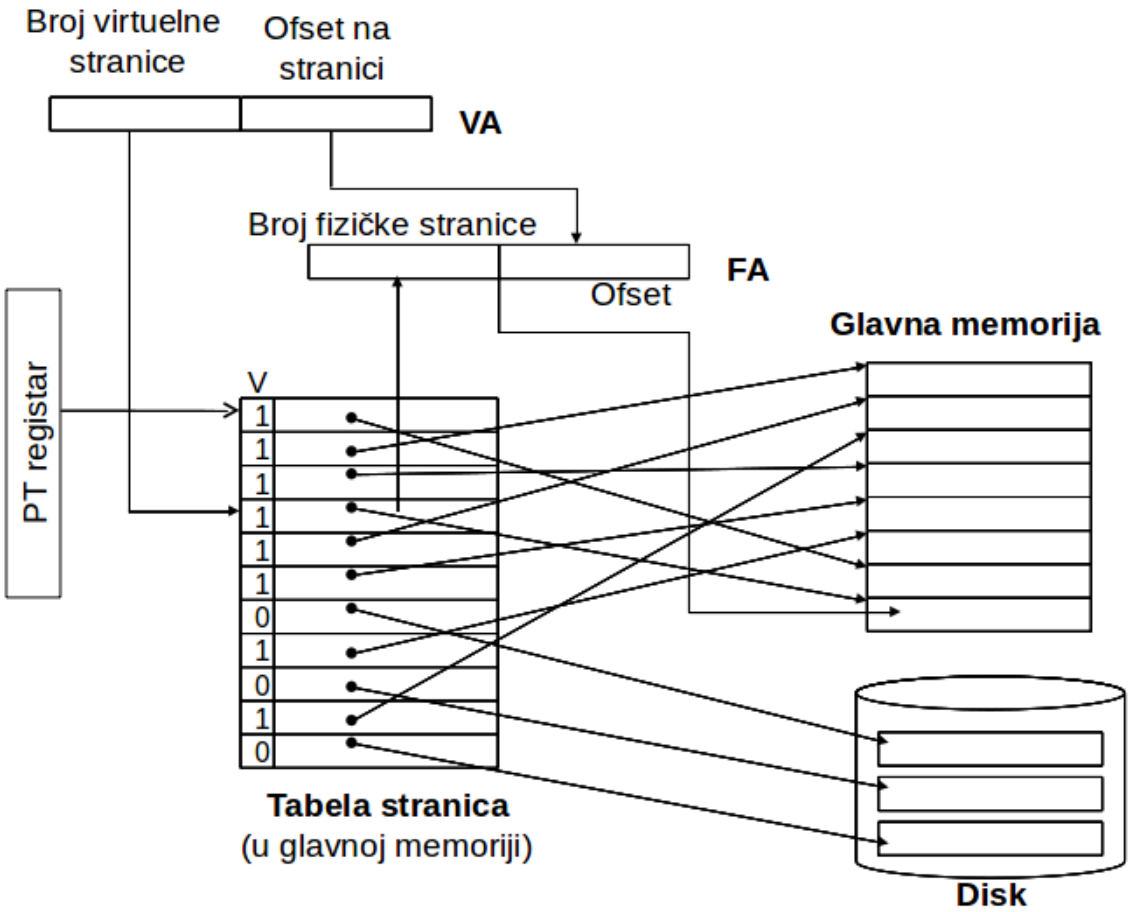


Slika 10.13. Prevođenje virtuelnih u fizičke adrese. Polje bita ofseta stranice se ne mijenjaju.

Prema oznakama na slici 10.13. karakteristike sistema virtuelne memorije su:

- 2^p – veličina stranice u bajtima,
- 2^n – veličina virtuelnog adresnog prostora i
- 2^m .- veličina fizičkog adresnog prostora.

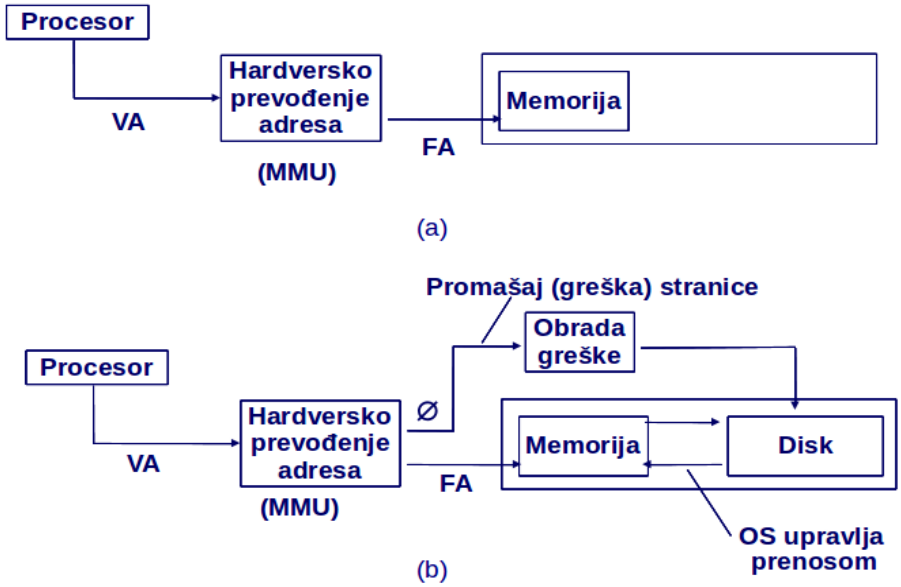
Detalniji mehanizam prevođenja virtuelnih u fizičke adrese je dat na slici 10.14. PT (od eng. Page table) registar procesora pokazuje na početak (bazu) tabele stranica aktivnog programa. Gornji (značajniji) dio viruelne adrese (broj virtuelne stranice) se sabira sa baznom adresom tabele stranica i pokazuje gdje se u tabeli nalazi informacija o rednom broju odgovarajuće fizičke stranice (na slici je to četvrti red odozgo u tabeli). Ako je u tom redu V bit postavljen, to znači da je tražena stranica u fizičkoj memoriji i da se u ostatku reda nalazi njoj odgovarajući redni broj stranice u memoriji. Na slici se vidi da pokazivač iz četvrtog reda tabele stranica pokazuje na zadnju stranicu glavne memorije. Preostali dio prevođenja se svodi na dodavanje (ne sbiranje već pridruživanje) donjih bita virtuelne adrese (ofset bajta/riječi unutar stranice) bitima rednog broja odgovarajuće fizičke stranice. Time je prevođenje završeno i fizička adresa, u primjeru na slici, pokazuje na lokaciju u memoriji koja je za ofset udaljena od početka zadnje fizičke stranice.



Slika 10.14. Mehanizam prevođenja adresa. PT registar pokazuje na početak tabele stranica aktivnog programa u memoriji.

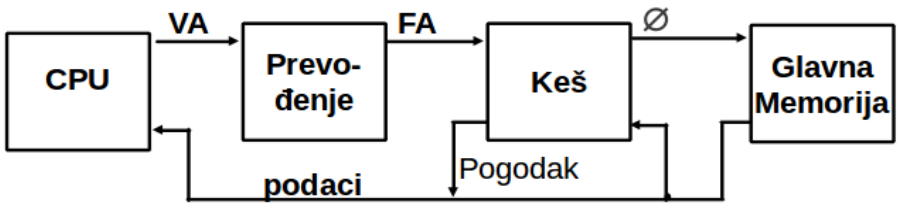
Iako je analogija virtuelne memorije sa kešom očigledna, postoje i značajne razlike. Prva je

razlika u brzini – vrijeme pristupa glavnoj memoriji je desetak puta veće (jedan red veličine) od vremena pristupa kešu. U virtuelnoj memoriji odnos vremena pristupa disku je 5 redova veličine veće od vremena pristupa memoriji. Ovo treba imati na umu prilikom određivanja politike keširanja stranica iz virtuelne u fizičku memoriju. Kod promašaja u kešu hardver mora razrješiti problem pribavljanjem traženog bloka iz memorije i nema vremena za promjenu konteksta (eng. **context switching**) - preuzimanje drugog programa (posla) u međuvremenu. Kod promašaja stranice u virtuelnoj memoriji (slika 10.15.b) nema smisla pustiti procesor da čeka dok operativni sistem traženu stranicu sa diska ne kopira u glavnu memoriju. Za to vrijeme (reda 10-tina milisekundi) procesor može izvršiti desetine miliona instrukcija – više nego dovoljno da promjenom konteksta može upravljati i softver. Programsko upravljanje i u ovom slučaju daje fleksibilnost u određivanju koju stranicu zamijeniti u memoriji.



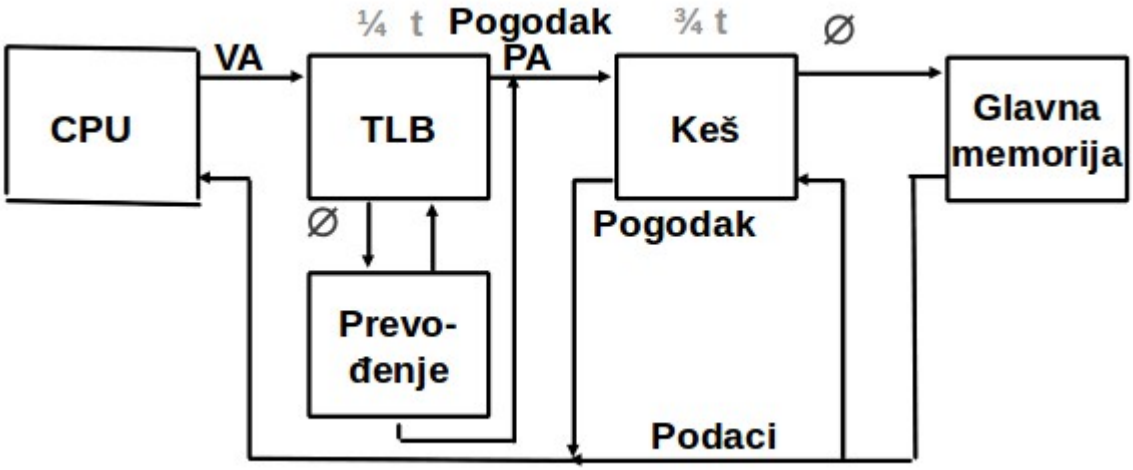
Slika 10.15. Prevođenje virtuelnih adresa sa pogotkom (a) i promašajem (b) u tabeli prevođenja

U dosadašnjem razmatranju mehanizma prevođenja adresa podrazumjevao se scenario kao na slici 10.16. Svako čitanje ili pisanje memorije je podrazumjevalo dva pristupa memoriji – jednog radi čitanja tabele stranica i prevođenja adresa, a drugog radi pristupa traženom podatku (u kešu ili memoriji). Ovakav pristup je vrlo spor i kao takav neprihvatljiv.



Slika 10.16. Prevođenje adresa pomoću tabele stranica u memoriji, svodi svako čitanje ili pisanje u memoriju na dva pristupa – jedan radi prevođenja adresa i drugi radi pristupa podacima – ovo bi bilo vrlo sporo i neprihvatljiv.

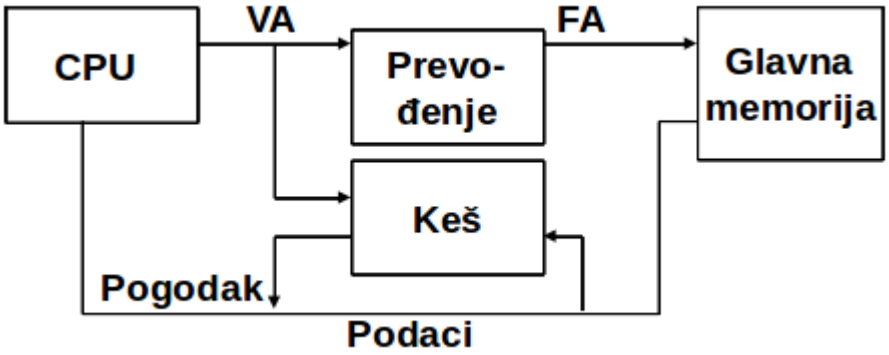
Najjednostavniji način ubrzanja prevođenja adresa se svodi na keširanje (dijela) tabele stranica. Na taj način se, pored keševa instrukcija i podataka, uvodi i keš adresa iz tabele stranica – bafer prevođenja adresa – **TLB** (eng. Translation Lookaside Buffer).



Slika 10.17. Bafer prevođenja adresa (TLB) u memorijskoj hijerarhiji

TLB je mali (jednostavni i brzi) hardverski keš smješten u dijelu procesora koji upravlja memorijom **MMU** (eng. Memory Management Unit) koji je prikazan na slici 10.15. Kako i kod pristupa adresama u memorijskom prostoru važe principi prostorne i vremenske lokalnosti, TLB ubrzava prevođenje broja virtuelne stranice u broj fizičke stranice, jer sadrži kompletne redove iz tabele stranica koji su najčešće korišteni u posljednje vrijeme. Ukoliko se obezbijedi da je vrijeme pogotka u TLB-u znatno kraće od vremena pogotka u kešu, a procenat pogodaka u oba je preko 95%, ovaj mehanizam značajno ubrzava pristup memoriji.

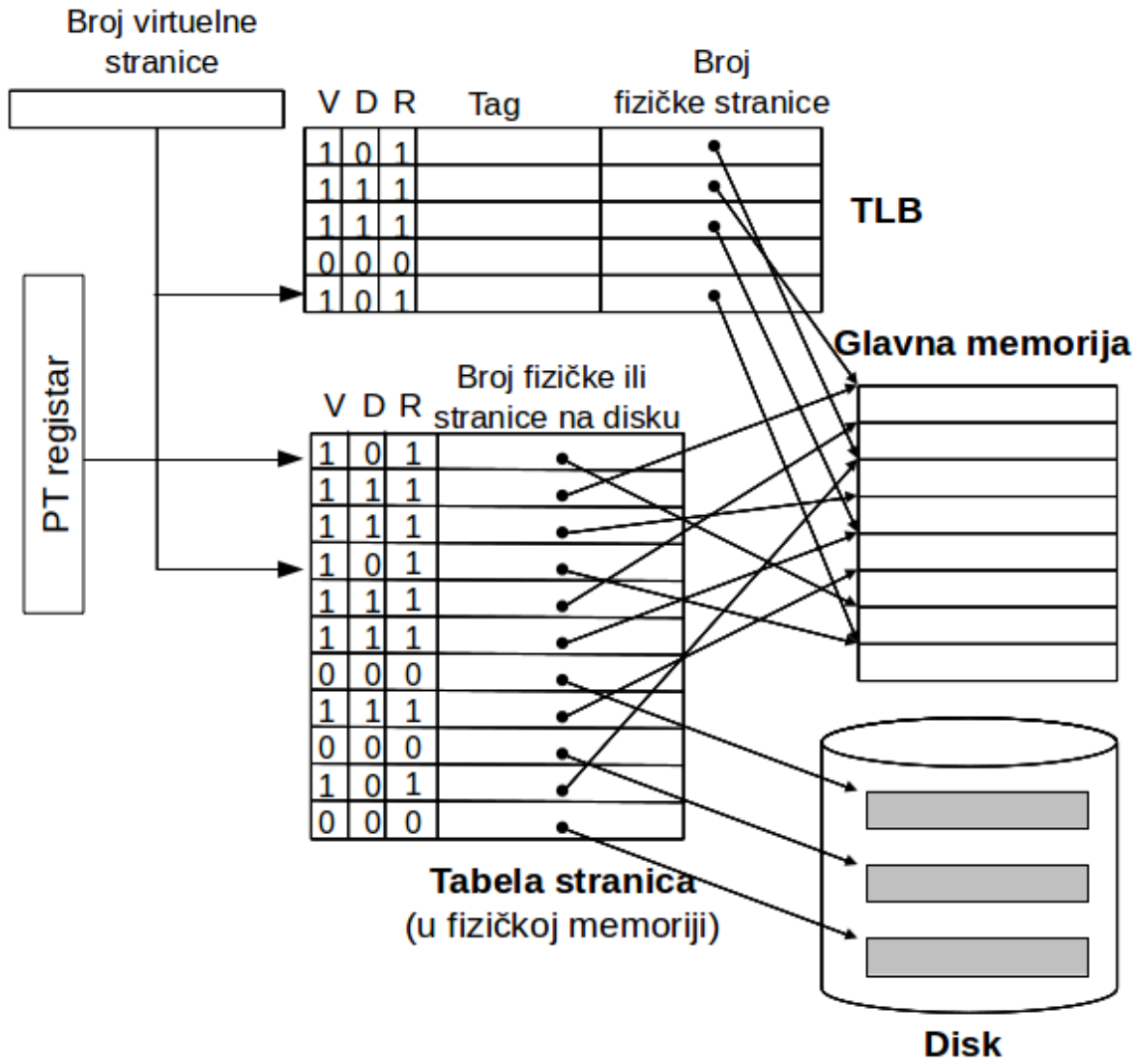
Ubrzani pristup glavnoj memoriji bi se mogao postići i keširanjem pomoću virtuelnih adresa (slika 10.18.). U ovom slučaju se prevođenje adresa vrši samo kod promašaja u kešu, radi njegovog osvježavanja. Ova tehnika bi bila vrlo efikasna da se u glavnoj memoriji ne nalaze dijelovi različitih programa koji se izvršavanju u svojim odvojenim virtuelnim adresnim prostorima.



Slika 10.18. Keširanje virtuelnim adresama - još jedan pokušaj ubrzavanja pristupa glavnoj memoriji u sistemu virtuelne memorije.

Ukoliko dva od tih programa dijele neku varijablu u memoriji, može se lako dogoditi da se ista varijabla kešira na dva mjesta u kešu (zbog različitih virtuelnih adresa) pa će svaki

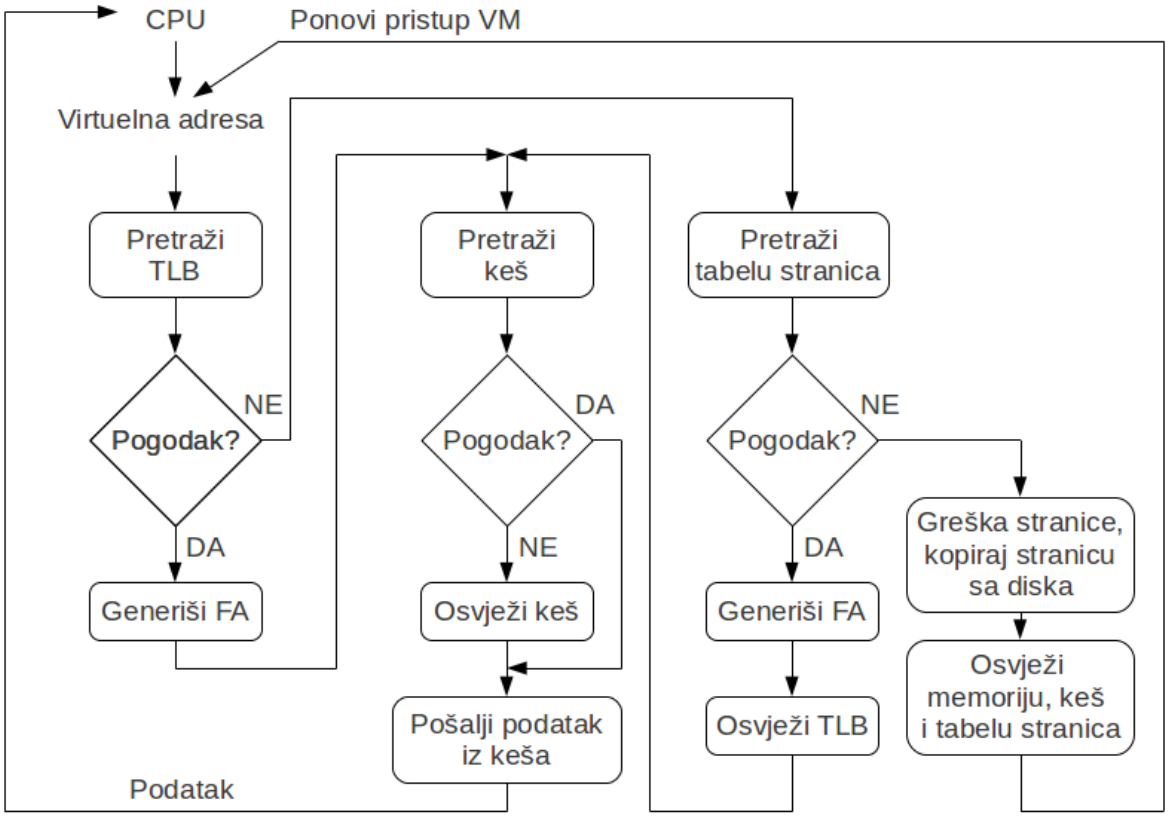
program osvježavati svoju kopiju, dok onaj drugi to neće moći znati. Ovakve dvostruke slike iste varijable (eng. **aliasing**) narušavaju konzistentnost izvršenja programa i ne mogu se tolerisati. Zbog toga se ova metoda ubrzanja pristupa memoriji ne koristi, a keširanje se isključivo vrši fizičkim adresama.
TLB, kao keš prevođenja, sadrži samo podatke o parovima brojeva virtuelnih i fizičkih stranica koje su prevođene u posljednje vrijeme (Slika 10.19.), pa se ne može desiti da se u njoj nalazi par čija je fizička stranica na disku.



Slika 10.19. Broj virtuelne stranice se poredi sa tagovima u TLB-u. U slučaju pogotka, broj fizičke stranice se dobije iz TLB-a, a u slučaju promašaja iz tabele stranica i to iz reda koji je za broj virtuelne adrese redova udaljen od početka tabele stranica. TLB pored keširanih parova rednih brojeva stranica koje se nalaze u fizičkoj memorji, sadrži i njihove statusne bite validnosti (V), izmjenjenosti (D) i korištenja (R).

Kod svakog pristupa memoriji, traži se emitovani broj virtuelne stranice u TLB-u. U slučaju pogotka, broj odgovarajuće fizičke stranice se koristi za generisanje fizičke adrese, a njen bit referenciranja R (od eng. **Reference**) se postavlja. Ukoliko se radi o pristupu memoriji radi pisanja, postavlja se i odgovarajući bit promjene stranice D (od eng. **Dirty**). U slučaju

promašaja, potrebno je odrediti da li se radi o promašaju-grešci u memoriji ili samo promašaju u TLB-u. Ako se tražena stranica nalazi u memoriji, promašaj u TLB-u znači da je potrebno keširati novi prevod iz tabele stranica, i ponoviti pristup memoriji (Slika 10.20.). Ukoliko stranica nije u memoriji, promašaj u TLB-u indicira stvarnu grešku-promašaj stranice, pa je potrebno angažovati operativni sistem pomoću odgovarajućeg izuzetka. Tipično, TLB sadrži od 16 do 512 linija [1], što je mnogo manje od broja stranica u glavnoj memoriji, pa su promašaji (samo) u TLB-u mnogo češći od stvarnih promašaja-greške stranice u memoriji. Nakon promašaja u TLB-u i dobavljanja traženog prevoda iz tabele stranica, treba odrediti par u TLB-u koji će se zamijeniti. Statusni biti odabranog para se moraju kopirati nazad u tabelu stranica jer su to jedine informacije koje su se mogle promijeniti od trenutka njihovog keširanja. Kako je procenat promašaja u TLB-u vrlo nizak, pristup "pisanja unazad" (eng. **write-back**) je vrlo efikasan.



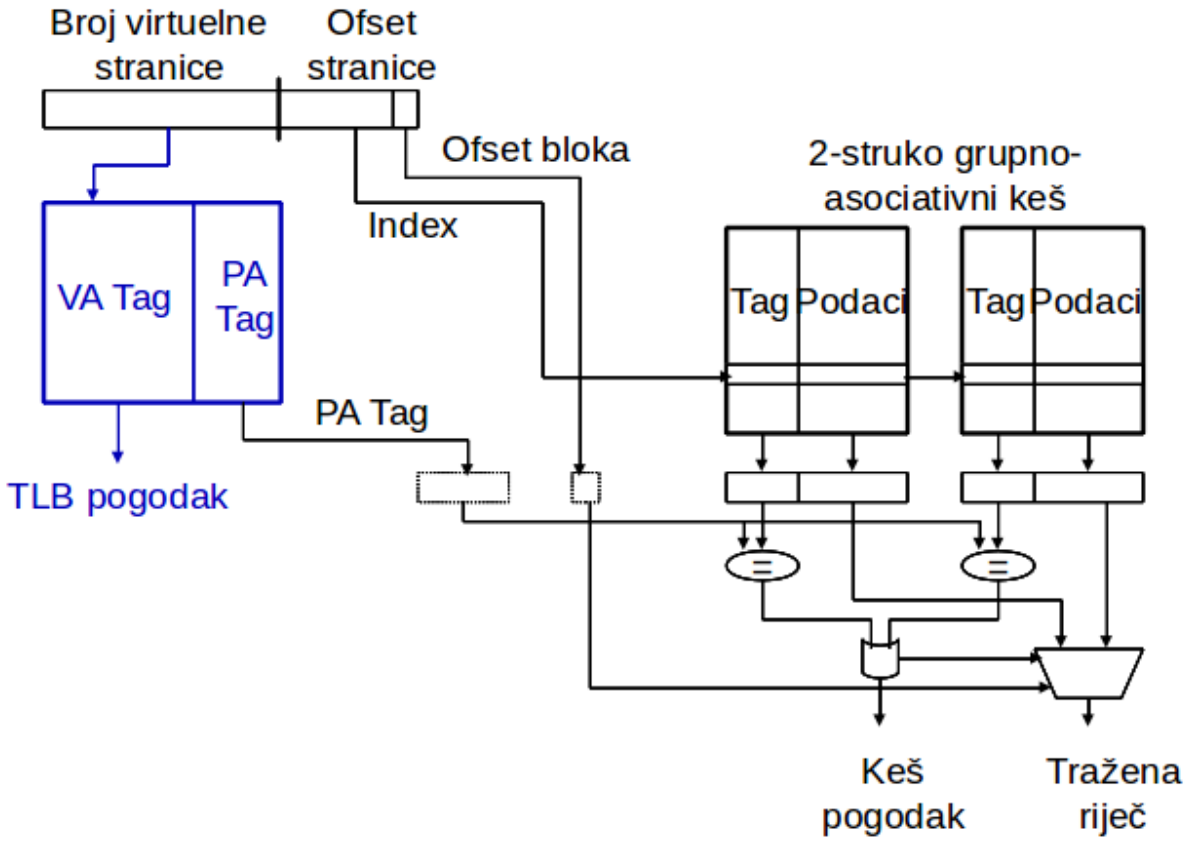
Slika 10.20.Dijagram toka mogućih događaja prilikom pristupa virtuelnoj memorji. Vidi se interakcije između keša, TLB-a i strananja u virtuelnoj memoriji. Memorijska hijerarhija se aktivira samo ako se izvodi instrukcija kojom se pristupa memoriji. Adresa se prvo provjerava u TLB-u da se vidi da li je prevod iz virtualne u fizičku stranicu u skoroj prošlosti urađen. Ako jeste i još uvijek je validan, prevedena adresa se čita iz TLB-a i koristi za pristup kešu. Ako nije, prevod se uradi pomoću tabele stranica i prevod se upisuje u TLB. Ovo se može izvesti brzo, jer se tablice stranica kešira i pristupati se može TLB-u i kešu istovremeno, u očekivanju pogotka u TLB-u. Ako je pretraživanje TLB-a neuspješno, keš pristup se ignoriše-prekida. Dešavaju se promašaji i u ostalim nivoima keša. Traženi podatak se mora naći u glavnoj memoriji, a cijeli blok upisati u keševe obrnutim redom od promašaja. Promašaj stranice nije moguć jer je prevod u TLB-u za tu stranicu bio validan.

Potpuno asocijativni TLB smanjuje procenat promašaja pa se, za male TLB-ove, ovo preslikavanje često koristi. Za veće TLB-ove traženje **LRU** para (najmanje korištenog u

zadnje vrijeme) za zamjenu može biti složeno, a hardver komplikovan. Ove operacije moraju biti vrlo brze i efikasne, pa korištenje softvera je u takvoj situaciji neprihvarljivo sporo. Pseudo-LRU algoritmi kao i oni koji se oslanjaju na R-bite, pa slučajnim odabirom među parovima kod kojih taj bit nije postavljen, jednostavnije i brže se određuje red u TLB-u za zamjenu.
Virtuelna memorija i keš čine jedinstvenu hijerarhiju, pa podatak ne može biti u kešu ako nije u glavnoj memoriji. U održavanju takve koherentnosti pomaže operativni sistem time što iz TLB-a briše podatke o stranicama koje odluči prebaciti na disk, a modifikuje tabelu stranica tako da pokušaj pristupa stranici koja je na disku izazove grešku-promašaj stranice u memoriji.
U najboljem slučaju sa slike 9.21., virtuelna adresa se prevodi u TLB-u a generisanom fizičkom adresom se traženi podatak nađe (pogodak) u kešu i šalje procesoru. U najgorem slučaju, pristup virtuelnoj memorji može izazvati promašaj u sve tri komponente memorijske hijerarhije – TLB-u, kešu i tabeli stranica. Ove i ostale kombinacija mogućih događaja su prikazane na slici 9.21. a detaljniji opis mogućnosti nastanka i uslova pod kojim se mogu desiti, su dati u tabeli 10.2.

Pogodak u:			Mogućnost i uslovi nastanka
TLB-u	Tabeli stranica	Kešu	
Da	Da	Da	Da! Idealan slučaj – bolje ne može! Tabeli stranica se ne pristupa nakon pogotka u TLB-u.
Da	Da	Ne	Da! Tabeli stranica se ne pristupa nakon pogotka u TLB-u.
Ne	Da	Da	Da! Promašaj u TLB-u, a fizička adresa u tabeli stranica
Ne	Da	Ne	Da! Promašaj u TLB-u, a fizička adresa u tabeli stranica, promašaj u kešu
Ne	Ne	Ne	Da! Promašaj/greška stranice
Da	Ne	Da ili Ne	Ne! Nemoguće prevođenje (pogodak u TLB-u) ako stranica nije u memoriji (tabeli stranica)
Ne	Ne	Da	Ne! Podaci ne mogu biti u kešu ako stranica nije u memoriji.

Tabela 10.2. Opis mogućnosti i uslova nastanka pogodaka i promašaja u sva tri komponente memorijske hijerarhije - TLB-a, keša i tabele stranica.



Slika 10.21. Preklapanje vremena pristupa TLB-u i kešu sa ciljem skraćenja vremena prevođenja adresa i pristupa kešu.

Vrlo efikasna tehnika za "sakrivanje" vremena prevođenja rednih brojeva virtuelne u redni broj fizičke stranice je preklapanje pristupa TLB-u i kešu (slika 10.21.). Ovom paralelizacijom se postiže skraćenje ukupnog vremena prevođenja i pretraživanja keša tako što se gornji dio polja bita virtuelne adrese – broj virtuelne stranice koristi za pristup TLB-u, dok se istovremeno donjim dijelom – ofsetom unutar stranice i bloka, indeksira keš. Dok traje pretraživanje TLB-a, traje i indeksiranje keša, nakon čega se porede tagovi iz izabranog reda u kešu porede sa brojem fizičke stranice iz TLB-a. U slučaju pogotka, biti ofseta unutar bloka (najniži biti polja bita adrese riječi u fizičke adrese) se koriste za izdvajanje tražene riječi iz pogođenog bloka u kešu.

10.4. Umjesto zaključka o memorijskoj hijerarhiji

Stranice virtuelne memorije (od 4KB do 64KB) su za tri reda veličine veće u odnosu na keš blokove , koji su obično veličine nekoliko riječi. Kopije najčešće korištenih blokova se čuvaju u kešu, kao i u glavnoj memoriji, a također i u slici virtualne memorije koja je pohranjena na disku. Kada se pristupa memoriji, prvo se pristupa TLB-u, a zatim kešu gdje se u slučaju pogotka riječ i nađe. Ako tražena riječ nije u kešu, blok u kome je tražena riječ se učita u keš iz glavne memorije, odakle se prosljeđuje procesoru. Ako se stranica koja sadrži riječ nije u glavnoj memoriji, onda se stranica kopira u glavnu memoriju sa diska, a blok se zatim kešira i šalje procesoru.
Korištenje virtualne memorije može dovesti do složene interakcije sa kešom. Na primjer, više

od jednog programa može koristiti keš i virtualnu memoriju, pa vremena izvršenja programa u dva prolaza (sa istim skupom podataka) mogu biti različita. Isto tako, kada modifikovani blok iz keša treba biti upisan natrag na glavnu memoriju, moguće je da je odgovarajuća stranica zamjenjena. Tada bi trebalo ponovo učitati stranicau sa diska, kako bi se upisao modifikovani blok iz keša u svoju fizičku stranicu. U praksi, međutim, ova situacija ne može nastati jer je memorijska hijerarhija inkluzivna - ako se blok na nižem nivou ukloni, svi odgovarajući blokovi na višim nivoima se uklanjaju. Drugim riječima, kada se stranica zamijeni-briše, sve njeno u keševima i TLB-u se briše.

11. Ulazno-izlazni podsistem

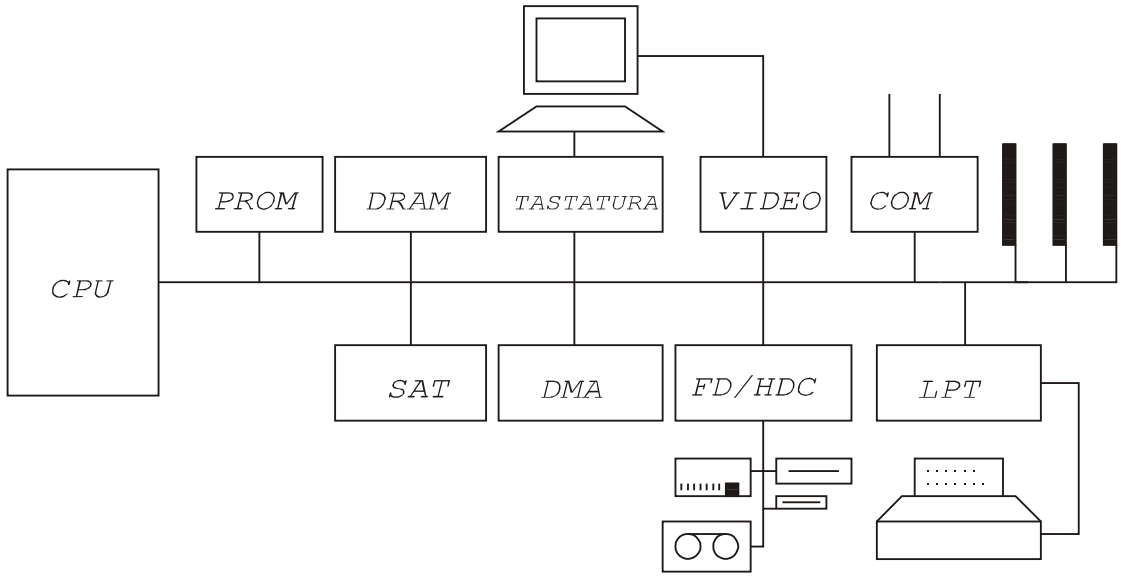
U prethodnim poglavljima je bilo riječi o procesoru kao o cetralnoj jedinici računara u kojoj se, kroz izvršavanje programa, vrši obrada podataka i memorijskom podsistemu u kome se nalaze programi i podaci koji se obrađuju. Digitalni računar komunicira sa vanjskim svijetom posredstvom ulazno-izlaznih (U/I) uređaja. Bez njih nijedna informacija nebi mogla niti ući niti izaći iz računara, a time ni računar nebi imao smisla. Performanse U/I uređaja se često zanemaruju, a u velikoj mjeri određuju performanse čitavog računara. Neka se neki program za mjerenje performansi računara (eng. benchmark) izvršava za 100 vremenskih jedinica, od čega se na U/I operacije troši samo 10. Neka je trend porasta performansi procesora takav da se one povećavaju 50% godišnje (Joy-ev zakon, važio do prvih nekoliko godina 21. vijeka). Tabela 11.1. prikazuje trend rasta performansi para procesor-memorija i njegov uticaj na ukupni trend ubrzanja čitavog računara sa nepromjenjenom brzinom U/I uređaja.

Godine	Vrijeme CPU	U/I vrijeme	Ukupno vrijeme	% U/I vrijemena
0	90	10	100	10
1	60	10	70	14
2	40	10	50	20
3	26.67	10	36.67	27
4	17.78	10	27.78	36
5	11.85	10	21.85	46
6	7.90	10	17.90	56
7	5.27	10	15.27	65
8	3.51	10	13.51	74
9	2.34	10	12.34	81
10	1.56	10	11.56	86

Tabela 11.1. Evolucija zaostajanja U/I uređaja za procesorom. Nakon 10 godina procesor je oko 58 puta brži, a čitav računar sa nepromjenjenom brzinom U/I uređaja samo oko 8,65 puta brži (100/11,56).

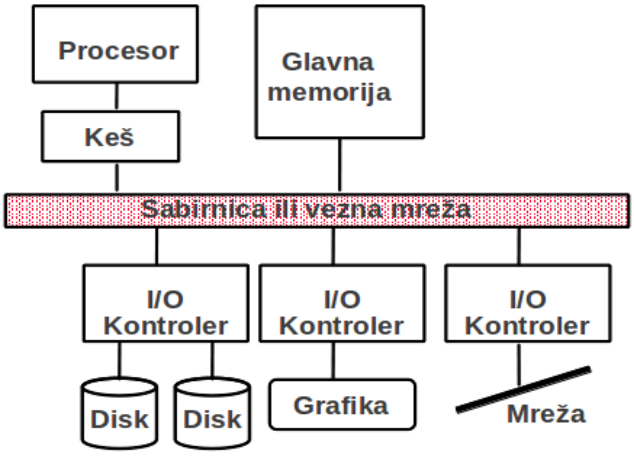
Iz tabele se vidi da stagniranje performansi U/I uređaja značajno ograničava ukupne performanse računara. U datom primjeru se pretpostavlja da je većina posla u programu vezana za rad procesora. U slučaju da se 90%vremena troši na U/I uređaje, ukupno ubrzanje računara nakon 10 godina bi bilo samo oko 11%. U programima koji više koriste U/I uređaje (npr. obrada transakcija u bazama podataka) napredak u performansama procesora manje utiče na ukupne performanse, a brzina U/I uređaja postaje odlučujuća. Performanse U/I uređaja se najčešće definišu kroz njihovu **propusnost** (količina podataka koju mogu prenijeti u jedinici vremena – npr KB/s, MB/s, ili u broju U/I operacija po sekundi) i **kašnjenje** (vrijeme odziva uređaja nakon traženja usluge). U različitim primjenama treba obratiti pažnju na različite kombinacije važnosti ovih parametara. Kod superračunara i naučnih proračuna kritična je propusnost podataka, i to najčešće izlazna u odnosu na ulaznu. Kod obrade transakcija u bankarstvu podjednako je važan broj transakcija u sekundi, kao i vrijeme odziva. Za fajl servere kritičan je broj fajlova koji se mogu obraditi (U/I operacija, čitatnja ili pisanja) u jedinici vremena.
Treba imati na umu da se kod mjerenja kapaciteta i propusnosti memorije 1KB definiše kao

1024 bajta (2^{10}), a 1MB kao 1024KB ili 1048576 bajta (2^{20}). Kod mjerenja propusnosti U/I uređaja 1KB/s je 1000B/s (10^3), a 1MB/s je 1000KB (10^6). Razlika nije velika, ali ipak postoji i sa prelaskom na GB i TB postaje veća.
Najčešće se, kao ulazni uređaji, koriste tastatura, miš i razni senzori, a kao izlazi ekran i štampač. Pored navedenih, tu spadaju i magnetni i optički podsistemi za masovni smještaj podataka (diskovi, trake) kao i razni komunikacioni kontroleri. Arhitektura prve generacije personalnih računara je prikiazana na slici 11.1.



Slika 11.1. Arhitektura prve generacije personalnih računara - komponente i njihova povezanost

Svi ulazno-izlazni uređaji su, preko svojih kontrolera, vezani na sabirnicu računara ili neku drugu veznu mrežu i komuniciraju sa procesorom i memorijom. Detaljniji prikaz takve strukture je dat na slici 11.2.



Slika 11.2. Povezivanje perifernih uređaja sa ostatkom računara preko odgovarajućih kontrolera

Svaki periferni (U/I) uređaj ima svoj kontroler, s tim što neki kontroleri mogu opsluživati više uređaja (npr. kotroler diskova). Periferni uređaji pretvaraju informacije iz jedne u drugu formu. Magnetni diskovi pretvaraju nule i jedinice u magnetni zapis (i obratno), dok štampač pretvata bajte u odštampana slova ili tačke u grafičkom prikazu. Različiti U/I uređaji imaju različite propusnosti. Tabela 11.2. prikazuje karakteristike nekih uređaja, kao i njihovu propusnost (orjentaciono).

Uređaj	Uloga	Korisnik	Količina podataka (KB/sek)
tastatura	ulaz	Čovjek	0.01
miš	ulaz	Čovjek	0.03
linijski štampač	izlaz	Čovjek	1
floppy disk	U/I	Mašina	50
laserski štampač	izlaz	Čovjek	200
optički disk	U/I	Mašina	5000
magnetni disk	U/I	Mašina	70000
grafički kontroler	izlaz	Čovjek	120000

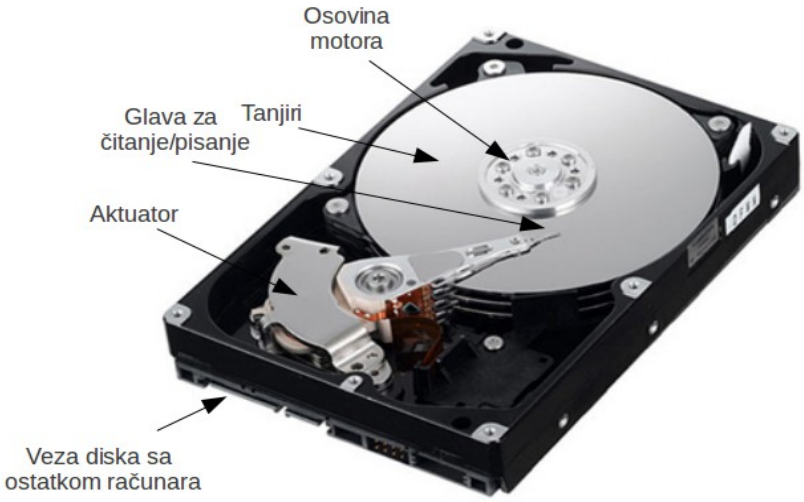
Tabela 11.2. Različite uloge i propusnosti pojedinih U/I uređaja

Prema tabeli mrežni kontroler obrađuje sto miliona puta (osam redova veličine) više podataka od kontrolera tastature. Pri tome, neki ulazni uređaji nisu u stanju poslati podatke onom brzinom kojom ih procesor čita (npr. čeka se pritisak na tastaturu ili pomak miša). Neki izlazni uređaji nisu u stanju prihvatiti podatke onom brzinom kojom ih procesor upisuje. Procesor treba da upravlja radom svih tih uređaja. Njegova komunikacija sa svakim uređajem se odvija preko kontrolera tog uređaja i to, najšešće, posredstvom dvaju veznih registara tog kontrolera. To su:

- kontrolni registar i
- registar podataka

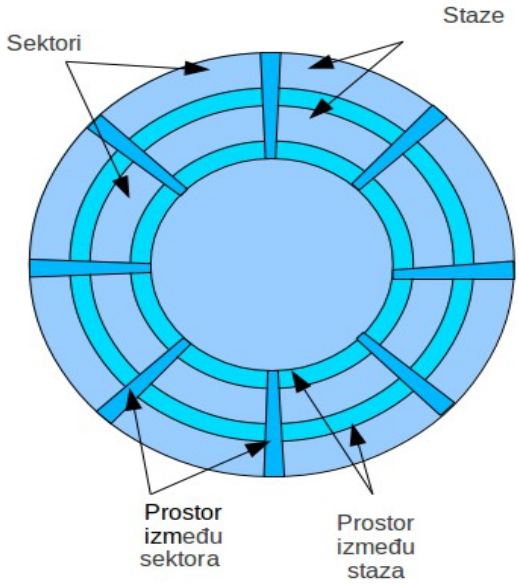
Kontrolni registar sadrži informacije o spremnosti uređaja za upis ili čitanje podataka. Registar podataka služi za upis ili čitanje podataka, nakon što kontrolni registar signalizira da je uređaj spreman.

Unutrašnja struktura jednog od najčešće korištenih U/I uređaja, tvrdog diska (eng. HDD od Hard Disk Drive) je data na slici 11.3.



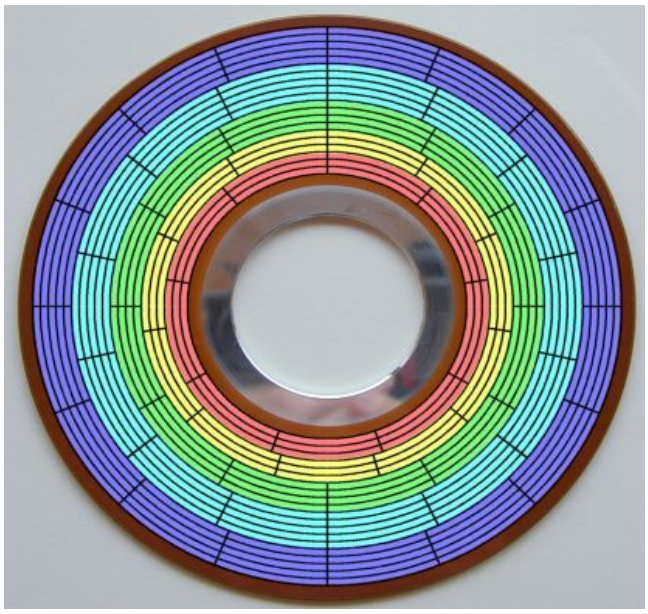
Slika 11.3. Komponente i organizacija tvrdog diska.

Kao što se vidi na slici 11.3., disk se sastoji od jednog ili više magnetnih medija (tanjira) koji se okreću na zajedničkoj osovini sa motorom diska. Disk na slici ima tri tanjira i šest glava za čitanje i pisanje magnetnih zapisa na magnetnim površinama sa obje strane tanjira. Biti se zapisuju pomoću polariteta magnetizacije na magnetnom mediju. Sve glave za čitanje/pisanje su pričvršćene na jednu ruku – polugu i kreću se zajeno polukružnim pokretima ruke, pri čemu se glave kreću između oboda i centra tanjira. Zapisi su organizovani po gusto zbijenim (do 50000 po strani tanjira) koncentričnim krugovima – **stazama** (eng. track). Staze su podijeljene na **sektore** (do 500 sektora po stazi, svaki po 512B podataka) koji čine osnovnu jedinicu informacije na disku. Sve staze, na svim diskovima, koje su podjednako udaljene od sredine diska čine jedan **cilindar**. Kada je bilo koja glava podešena na određenu stazu, čitanje ili pisanje po ostalim stazama istog cilindra se može obaviti bez pomjeranja glava. Pojednostavljeni prikaz organizacije staza i sektora na jednoj površini tanjira je prikazan na slici 11.4. Staze su, kao i sektori, međusobno odvojeni. Razmaci među stazama obezbjeđuje izolaciju susjednih megnetnih zapisa, dok se razmaci među sektorima koriste za razdvajanje osnovnih jedinica informacija (sadržaja sektora).



Slika 11.4. Staze i sektori na tanjiru diska
- organizacija i raspored

Svaki sektor sadrži istu količinu informacija, pa je lako uočiti da je gustoća zapisa najveća na unutršnjoj stazi, a opada kada je staza bliža vanjskoj. Zato se pribjegava varijabilnom raspoređivanju sektora po zonama koje čine grupisane susjedne staze sa istim rasporedom sektora. Slika 11.5. prikazuje jednu stranu tanjira diska na kome varira broj sektora po stazi u zavisnosti u kojoj od 5 zona se ona nalazi. U ovom slučaju je i broj staza neravnomjerno raspoređen po zonama. Na ovaj način je povećan kapacitet diska ali je brzina čitanja ili pisanja različita od zone do zone, što zahtjeva složenije mehanizme upravljanja.



Slika 11.5. Disk sa varijabilnim brojem sektora po stazi. 20 staza je svrstano u 5 zona različitih boja.

Tanjiri u disku se okreću konstantnom brzinom, koja je definisana određenim standardima. U tabeli 11.3. su date standardne brzine okretanja medija.

Diskovi brzine 3600 i 4800 obrtaja u minuti su bili popularni 80-ih i 90-ih godina prošlog vijeka. Diskovi od 5400 i 7200 o/ min se najčešće koriste u prenosnim i kućnim računarima, dok se najbrži koriste u naprednim radnim stanicama, serverima i superračunarima. **Rotaciono kašnjenje** (ent. Rotational latency) je vrijeme potrebno da bi se disk/tanjir okrenuo za pola kruga. To je prosječno vrijeme čekanja da se traženi podatak nađe ispod glave diska koja je već postavljena na traženu stazu.

Brzina rotacije (o/min)	Prosječno kašnjenje (ms)
3600	8.33
4800	6.25
5400	5.56
7200	4.17
10000	3.00
15000	2.00

Tabela 11.3. Standardne brzine okretanja tanjira u diskovima i odgovarajuća rotaciona kašnjenja kod pristupa podacima.

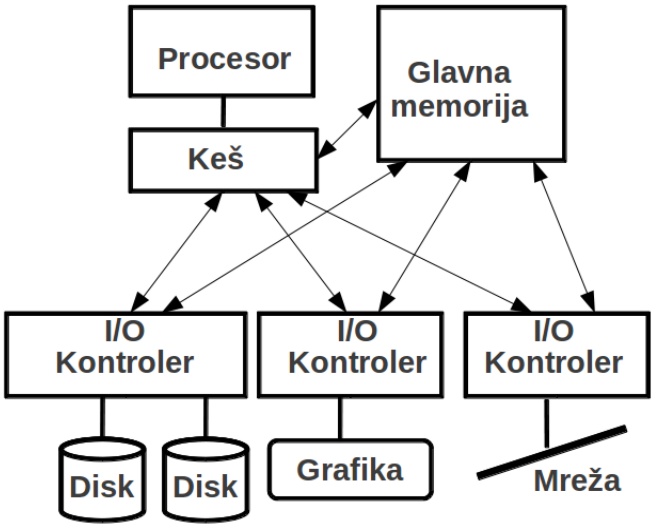
Brzina okretanja	**3600 RPM**	**5400 RPM**	**7200 RPM**	**10,000 RPM**	**15,000 RPM**
Oznaka diska	CDC Wrenl 94145-36	Seagate ST41600	Seagate ST15150	Seagate ST39102	Seagate ST373453
Godište	1983	1990	1994	1998	2003
Kapacitet (GB)	0.03	1.4	4.3	9.1	73.4
Širina diska (inča)	5.25	5.25	3.5	3.5	3.5
Prečnik tanjira (inča)	5.25	5.25	3.5	3	2.5
Interfejs	ST-412	SCSI	SCSI	SCSI	SCSI
Propusnost (MB/s)	0.6	4	9	24	86
Kašnjenje (ms)	48.3	17.1	12.7	8.8	5.7

Tabela 11.4. Pet generacija tvrdih diskova i njihovi osnovni podaci. Propusnost se odnosi na maksimalnu brzinu prenosa formatiranih podataka sa medija (ne iz keša na disku). Kašnjenje je zbir rotacionog kašnjenja i prosječnog vremena pozicioniranja glava (kašnjenje elektronike kontrolera na disku je relativno zanemarivo (ispod 0.2ms).

Ukupno vrijeme traženja željenog sektora na disku (eng. seek time) je zbir rotacionog kašnjenja tanjira i prosječnog vremena potrebnog da se glava postavi na željenu stazu. Vrijeme postavljanja glave zavisi od njenog prethodnog položaja – udaljenosti od željenog položaja i toga da li se želi čitati ili pisati. U slučaju čitanja, ono može početi i prije preciznog postavljanja glave nad sredinu staze (elektronika može filtrirati smetnje susjedne staze), dok kod pisanja postavljanje traje nešto duže jer je neophodno završiti precizno pozicioniranje prije početka pisanje. Tabela 11.4. prikazuje tehničke podatke pet diskova iz različitih generacija komercijalnih diskova namjenjenih profesionalnoj upotrebi. U njoj se vidi da su kašnjenja diskova (vremena traženja) uvećana u odnosu na, ranije pomenuta, rotaciona kašnjenja tanjira. Za razliku od kašnjenja koje dato kao statistička (prosječna) vrijednost, propusnost je data kao maksimalna brzina prenosa podataka od računara prema diskovima (mediju) i obratno. Stvarna brzina prenosa u realnim uslovima zavisi od niza faktora – položaju traženih staza i sektora, veličine i rasporeda dijelova datoteke koja se čita ili piše, popunjenosti diska itd. Savremeni diskovi obavezno sadrže interne bafere – keš memoriju kapaciteta reda desetine megabajta, za keširanje najčešće korištenih podataka u zadnje vrijeme. Zato je performanse diskova teško opisati sa nekoliko statističkih podataka. Mjerenje performansi praznog i popunjenog diska može dati potpuno različite rezultate.

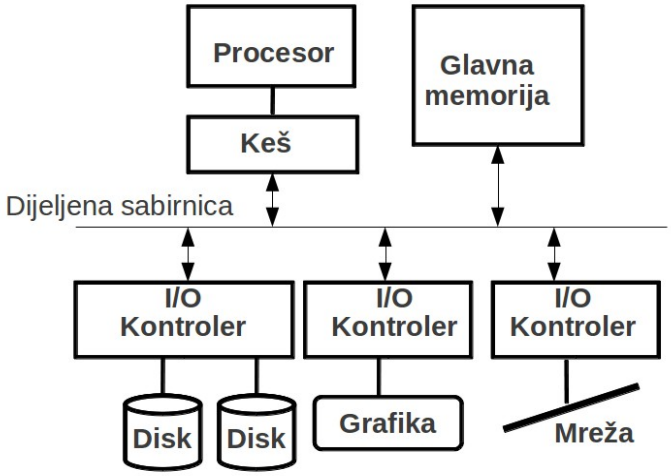
Ostali periferni uređaji, kao što su monitori/ekrani, grafičke kartice (kontroleri), skeneri, štampači i mrežne kartice, neće biti detaljno opisivani kao diskovi, već će biti apstrahovani svojim najvažnijim arhitekturalnim osobinama viđenim izvana (propusnosti i kašnjenje).

Povezivanje U/I uređaja sa ostatkom računara na slici 11.2. nije precizirano po pitanju karaktera vezne mreže koja povezuje komponente. Informacije (instrukcije i podaci) teku između memorije i procesora, sa jedne strane, i U/I uređaja sa druge, s tim što je moguća i direktna komunikacija između neka dva U/I uređaja. Postoje razne mogućnosti realizacije vezne mreže, od kojih je, na prvi pogled, najlogičniji ona kod koje su sve komponente međusobno povezane direktnim vezama “tačka-u-tačku”, kao na slici 11.6.



Slika 11.6. Povezivanje komponenti računara direktnim vezama "tačka-u-tačku".

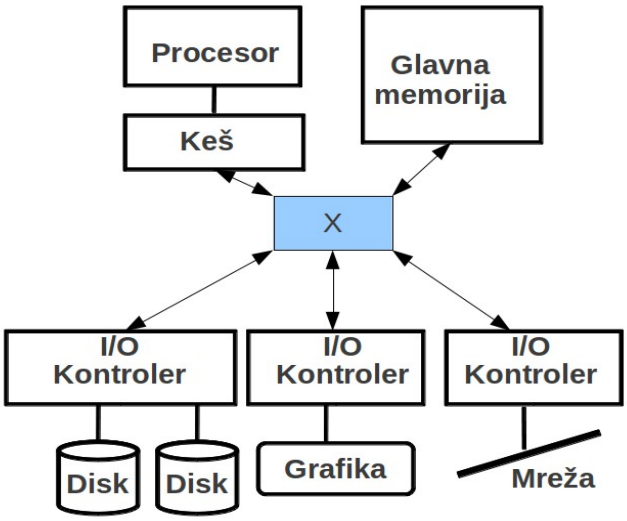
Na prikazani način svaka od komponenti bi mogla komunicirati sa svim ostalim u svakom trenutku. Svaki par komponenti, u ovom slučaju, ima svoj odvojeni spojni put – ostvarena je puna povezanost. Problem kod ovakvog povezivanja je da svaki učesnik u prenosu podataka, po pravilu, može upravljati samo jednom komunikacijom u jednom trenutku – ostali njegovi linkovi su neiskorišteni. Mnogo jednostavniji, ali i neefikasniji, način povezivanja bi bio pomoću zajedničke sabirnice – slika 11.7.



Slika 11.7. Povezivanje komponenti sistema preko dijeljene sabirnice.

Sabirnica je skup spojnih vodova (žica) koji čini dijeljeni komunikacioni kanal za prenos kontrolnih signala, adresa i podataka između bilo koje dvije komponente u sistemu. Inicijator prenosa (najčešće procesor) pomoću adresnih linija određuje s kim želi komunicirati. Upravljačkim signalima definiše smjer i vremenski slijed događaja u toku prenosa, dok se linijama podataka vrši razmjena podataka između izvorišta i odredišta. Glavna memorija nikada nije inicijator prenosa podataka na sabirnici, dok napredni U/I kontroleri to mogu biti. Ukoliko više parova želi komunicirati u istom trenutku, to se može obaviti samo kroz

sekvencu ciklusa prenosa podataka na sabirnici između pojedinih parova – nikada više od jednog para u jednom trenutku ne može biti aktivno na sabirnici. Uprkos ovim ograničenjima, povezivanje sabirnicom je vrlo jeftino i (zato) efikasno.
Istorija računarskih arhitektura je pokazala da sve dijeljene sabirnice vremenom postaju usko grlo u sistemu – sve je više potencijalnih učesnika i potrebna je sve veća brzina prenosa podataka uz minimalno kašnjenje.
Kompromis između pune povezanosti i dijeljene sabirnice predstavlja povezivanje komponenti preklopnikom (eng. switch) – usmjerivačem saobraćaja sličnim telefonskoj centrali (eng. crossbar ili X-bar) kao na slici 11.8.



Slika 11.8. Povezivanje komponenti računara preko preklopnika

Zadatak preklopnika, označenog na slici sa "X", da uspostavi veze između više od jednog para učesnika u saobraćaju u svakom trenutku. Time je povećan paralelizam u komunikaciji u odnosu na povezivanje dijeljenom sabirnicom, a pojednostavljeno komuniciranje svake komponente u odnosu na punu povezanost "tačka-u-tačku" – svaka komponenta ima jednu vezu sa ostatkom sistema.
Poređenje pomenutih načina povezivanja komponenti računara je dato u tabeli 11.5.

	Puna povezanost	Dijeljena sabirnica	Preklopnik
Cijena	Visoka	Niska	Visoka
Propusnost	Visoka	Niska	Visoka
Broj veza po komponenti	Više	Jedna	Jedna
Proširenje	Teško	Lako	Teško

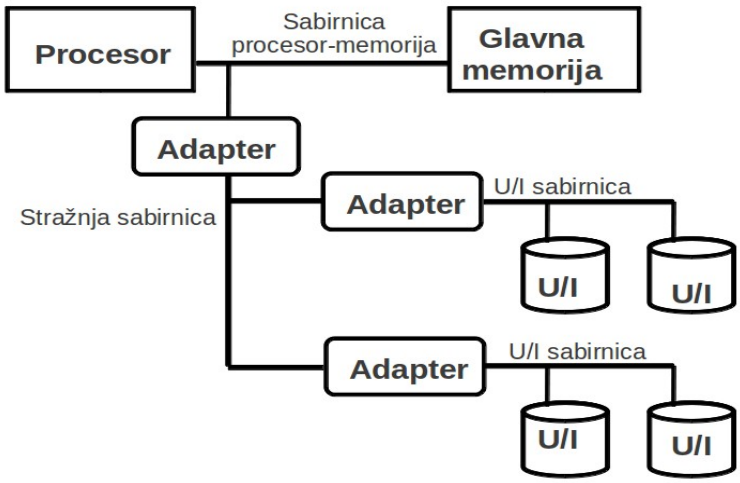
Tabela 11.5. Poređenje tri načina povezivanja komponenti računara

Osnovni nedostatak dijeljene sabirnice, niska propusnost, se može popraviti uvođenjem višestrukih sabirnica, tako da se više prenosa može odvijati paralelno. Time se povećavaju performanse ali i cijena izvedbe takvih veznih mreža. Sa druge strane, visoka cijena preklopnika se može smanjiti korištenjem više jednostavnijih preklopnika. Neke komponente u sistemu međusobno ne komuniciraju (npr. grafički podsistem sa mrežnim ili sa diskovima) pa se preklopnik može pojednostaviti bez gubitka performansi sistema.

Po namjeni, postoje tri osnovna tipa sabirnica:

1. sabirnice za povezivanje para procesor-memorija,
2. sabirnice za povezivanje ulazno-izlaznih uređaja I
3. stražnje sabirnice (eng. backplane).

Sabirnice koje povezuju procesore (i u njima keš memorije) i memorije moraju biti vrlo brze sabirnice i obično su fizički vrlo kratke (što smanjuje kašnjenje). Prilagođene su memorijskom sistemu da bi se postigla maksimalna propusnost veze sa procesorom i to optimizacijom prenosa keš blokova u rafalnom režimu (eng. burst mode). Svaki proizvođač procesora definiše sabirnicu za vezu sa memorijom, pa se ona smatra nestandardnom (eng. proprietary). Na toj sabirnici se nalazi i vezni adapter – most (eng. bridge) za vezu sa U/I sabirnicom – slika 11.9.

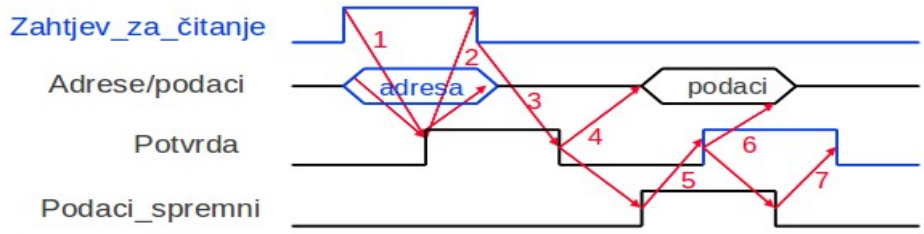


Slika 11.9. Računarski sistem sa tri vrste sabirnica. Adapteri odvajaju i prilagođavaju saobraćaje na različitim sabirnicama.

U/I sabirnice imaju zadatak da povežu U/I uređaje sa ostatkom sistema i da, pri tome, prilagode različite brzine prenosa različitih uređaja brzini sabirnice na koju je vezana. Ove sabirnice su duže i sporije od sabirnica na koje su vezane (procesor-memorija ili stražnje). Da bi povezale različite U/I uređaje različitih proizvođača, one moraju biti realizovane po određenom standardu – definisanom broju linija, protokolu, brzini prenosa itd. Primjeri industrijskih standarda za ovakve sabirnice su SCSI, USB i Firewire.

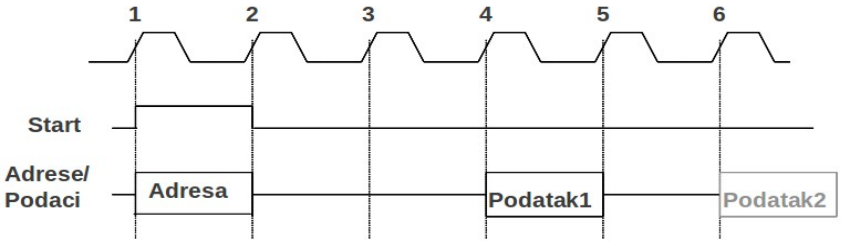
Stražnje sabirnice su dobile ime u doba kada su se računari realizovali od komponenti na odvojenim štampanim pločama, koje su se, preko konektora na svojoj ivici, povezivale u sistem preko “stražnje ploče” - sabirnice na stražnjoj strani ormara ili nekog drugog kućišta računara. Na taj način su se povezivale sve komponente računara – procesor sa memorijom i U/I uređajima, a sa prednje strane ploča su bili lampice i prekidači (korisnički interfejs). Ovakve sabirnice su bile ili standadizovane ili nestandadne (interni standardi proizvođača) i sve se rjeđe koriste. Na slici 11.9. je prikazana kao veza između najbrže (procesor-memorija) i najsporije (U/I) sabirnice. Primjeri industrijskih standarda za ovakve sabirnice su ATA i PCIexpress. Matične ploče savremenih personalnih računara sa konektorima za U/I uređaje, predstavljaju moderne verzije stražnjih sabirnica, samo što su procesor(i) i memorije odvojeno povezani na takvim pločama, a i U/I uređaji ne dijele sabirnicu već su povezani preko više adaptera/mostova i preklopnika.

Prenos informacija po sabirnici može biti organizovan na dva načina – asinhrono i sinhrono. Kod asinhronog prenosa, signal sata nije uključen u upravljački mehanizam prenosa – slika 11.10.



Slika 11.10. Vremenski dijagram asinhronog ciklusa čitanja iz memorije. Za prenos adrese i podatka koristi se ista sabirnica vremenski multipleksirana.

Strane, učesnice u prenosu, jedna drugoj signaliziraju šta treba uraditi sljedeće. Inicijator prenosa (procesor ili U/I uređaj) aktivira signal “Zahtjev_za_čitanje” i time daje na znanje memoriji da pokreće ciklus čitanja na sabirnici. Istovremeno, inicijator na jedinstvenoj sabirnici za adrese i podatke, objavljuje adresu u memoriji sa koje traži podatak (riječ). Memorija, nakon nekog vremena (svojom maksimalnom brzinom), aktivira signal “Potvrda” i na taj način signalizira da je primila zahtjev i zapamtila adresu (strelica 1). Inicijator nakon aktiviranja signala “Potvrda” zna da može što prije deaktivirati “Zahtjev_za_čitanje” i ukloniti adresu sa sabirnice (strelica 2). Ovaj ciklus “dogovaranja” (eng. handshake – rukovanje) se završava tako što memorija deaktivira signal “Potvrda” (strelica 3) i, nakon četiri faze, slijedi drugi ciklus dogovaranja oko prenosa podataka. Kada memorija, svojom maksimalnom brzinom, obradi traženi zahtjev za čitanje i podaci postanu raspoloživi, oni se prikazuju na sabirnici za adrese i podatke i memorija aktivira signal “Podaci _spremni” (strelica 4). To je znak inicijatoru da su podaci spremni, nakon čega on signalizira njihovo preuzimanje aktiviranjem signala “Potvrda” (strelica 5). Memorija, nakon toga, najbrže što može uklanja podatke sa sabirnice i to signalizira deaktiviranjem signala “Podaci_spremni” (strelica 6). Zadnju fazu dogovaranja oko prenosa podataka predstavlja deaktiviranje signala “Potvrda” (strelica 7) od strane inicijatora, čime je dat signal memoriji da je prenos završen i da je sabirnica spremna za neki drugi prenos. Prenos podataka je izveden kroz četiri faze dogovaranja prikazana na slici strelicama 4, 5, 6 i 7. Trajanje prenosa zavisi samo od brzina kojom učesnici u prenosu mogu da izvedu pomenutu sekvencu događaja i ne zavisi od nekog vanjskog signala sata.
Protokol na sinhronim sabirnicama je zasnovan na sasvim drugim principima. Svi događaji su usklađeni sa sinhronizacionim signalom sata (slika 11.11)



Slika 11.11. Vremenski dijagram prenosa na sinhronoj sabirnici.

U računarskim sitemim sa više odvojenih sinhronih sabirnica, svaka može imati svoj odvojen sinhronizacioni signal sata i oni mogu biti različitih frekvencija, zavisno od propusnosti podataka koja se od njih zahtjeva. Svi događaji prilikom čitanja iz memorije na sinhronoj sabirnici, na slici 11.11., su vremenski određeni signalom sata i svaki učesnik u prenosu tačno zna šta i kada treba da uradi. Neka se svi događaji odnose na uzlazne ivice signala sata. Na početku ciklusa prenosa inicijator mora signalizirati početak (signal "Start") i istovremeno saopštiti adresu memorijske lokacije kojoj želi pristupiti radi čitanja. Trajanje signala "Start" može biti jedan ili više (cijeli broj) ciklusa sata. Nakon što je memoriji saopštena adresa riječi koju inicijator želi čitati, od nje se očekuje da, nakon određenog broja ciklusa sata, prikaže traženi sadržaj na sabirnici. Na slici se to dešava dva ciklusa nakon deaktiviranja "Start" signala. Nema dogovaranja o tome da li je memorija primila adresu, kao ni o brzini pristupa podacima. Podrazumjeva se da će podaci biti na sabirnici nakon uzlazne ivice signala sata broj 4. Svi ostali učesnici u saobraćaju na ovakvoj sabirnici moraju znati ovaj protokol (šta se dešava u kojem ciklusu sata) i po njemu se ponašati. Zato je ovaj vremenski dijagram jednostavniji od onoga kod asinhrone sabirnice jer nema signala za dogovaranje.
Propusnost sabirnica je određena širinom sabirnice – brojem bita koji se prenose odjednom, brzinom prenosa jedne riječi, kao i mogućnošću prenosa bloka riječi.
Propusnost asinhronih sabirnica zavisi od brzine dogovaranja učesnika u prenosu, jer svi rade svojom maksimalnom brzinom. Brži učesnici ranije završe svoj prenos. Kod sinhronih sabirnica propusnost je unaprijed određena vremenskim dijagramom protokola prenosa i ne mijenja se. Svi učesnici obavljaju prenos istom brzinom. Postoje i sinhrone sabirnice kod kojih sporiji uređaji imaju mogućnost signalizirati inicijatoru kada su spremne za prenos, ubacivanjem cijelog broja ciklusa čekanja prije ciklusa planiranog za prenos podataka.
Da bi se propusnos sabirnica povećala, a iskoristile prednosti savremenih memorijskih sistema koji podržavaju strananje (eng. paging) i/ili preplitanje (eng. interleaving), kao i potrebe blokovskog načina (eng. burst mode) prenosa podataka, nakon pristupa prvoj riječi u memoriji, ostale susjedne riječi u bloku se mogu prenositi maksimalnom brzinom – nema potrebe da se eksplicitno adresiraju, a spremne su kada i prva riječ. Zato je na slici 11.11. osjenčeno prikazan i prvi naredni podatak – riječ iz bloka koji se čita.

Posljednjih godina dešava se tranzicija u arhitekturama sabirnica. Prelazi se sa paralelnih širokih sinhronih na asinhrone uske sabirnice. Refleksije u vodovima i različita kašnjenja signala (uključujući i signal sata) onemogućuju prenos po 32-bitnim ili 64-bitnim dvosmjernim sabirnicama, dužine 20-tak centimetara, punom brzinom na frekvencijama višim od oko 200MHz, ali je moguće preći na sabirnice sa manjim brojem jednosmjernih vodova, koji prenose podatke na frekvencijama od oko 2GHz. U tabeli 11.6 su data dva primjera takvih tranzicije standardnih sabirnica – PCI u PCIexpress i ATA u SerialATA.

	PCI	PCIexpress	ATA	Serial ATA
Ukupan broj žica	120	36	80	7
Broj žica za podatke	32 – 64 (2-smjera)	2 x 4 (1-smjer)	16 (2-smjera)	2 x 2 (1-smjer)
Sat (MHz)	33 – 133	635	50	150
Propusnost (MB/s)	128 – 1064	300	100	375 (3 Gbps)

Tabela 11.6. Karakteristike PCI i ATA sabirnica i njihovih nasljednika.

Propusnosti navedene u tabeli se odnose na maksimalne (vršne) propusnosti prilikom rafalnog prenosa spremnih blokova podataka.
Glavne metode povećanja propusnosti sabirnice su:

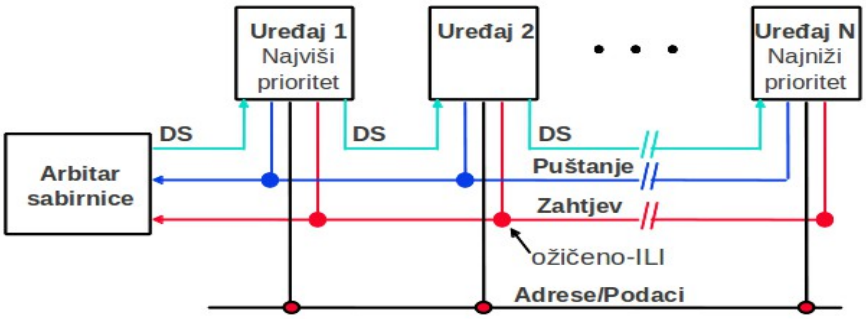
1. proširenje sabirnice (prenos više riječi u jednom trenutku),
2. odvajanje sabirnice adresa i sabirnice podataka,
3. korištenje sinhronih protokola (jednostavniji su i brži) i
4. prenos više riječi u jednom bloku (rafalni prenos).

Kod savremenih računarskih sistema, više komponenti - procesora, koprocesora i ulazno-izlaznih uređaja, mogu postati inicijatori – "vlasnici" sabirnice (eng. **master** - gazda), da bi mogli pisati ili čitati memoriju i međusobno komunicirati. Podčinjeni uređaji (eng. slave) su oni sa kojim inicijatori komuniciraju – odgovaraju na inicirani prenos na sabirnici (tipično memorija ali i U/I uređaji). U slučaju da procesor piše blok riječi u memoriju, on je inicijator a memorija je podčinjeni uređaj. Kada procesor instruira kontroler diska da izvrši određeni prenos iz memorije na disk ili obrnuto, procesor je inicijator, a kontroler diska podčinjeni uređaj. Kada kontroler diska želi da čita ili piše blok podataka u memoriji, on postaje inicijator prenosa. Memorija je uvijek podčinjeni uređaj.

Da se nebi desilo da dvije ili više komponenti u isto vrijeme pokušaju postati inicijatori prenosa i izazovu konflikt (logički, ali i električni) na sabirnici, neophodno je vršiti **arbitriranje na sabirnici**. Ono podrazumjeva odlučivanje ko u kom trenutku može postati inicijator prenosa i vlasnik sabirnice. Pri tome se može voditi računa o prioritetima uređaja ali i o tome da svi ravnopravni uređaji, na zahtjev, moraju dobiti pravo korištenja sabirnice. Arbitriranje može biti:

1. centralizovano – jednonivoski model, pomoću ulančavanja uređaja po prioritetu (eng. daisy chaining),
2. centralizovano paralelno,
3. distribuirano (decentralizovano) i
4. otkrivanjem kolizije na sabirnici (eng. collision detectrion).

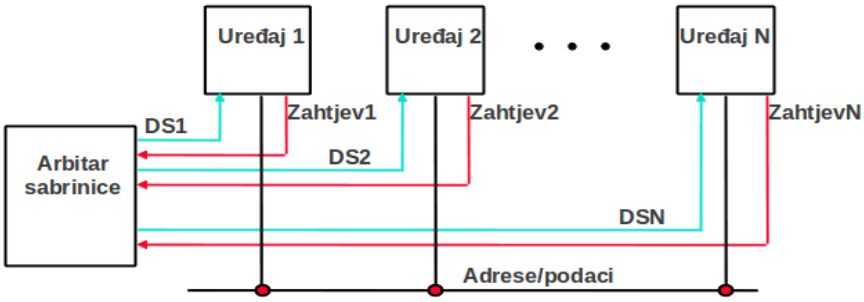
Jednonivoski centralizovani mehanizam arbitriranja pomoću ulančavanja uređaja po prioritetu je prikazan na slici 11.12.



Slika 11.12. Centralizovano arbitriranje - jednonivoski model

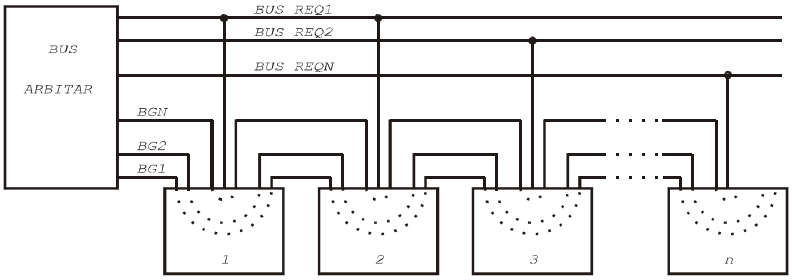
Signali “Zahtjev” i “Puštanje” su ožičeno-ILI strukture kojima mogu upravljati svi U/I uređaji. Signalom “Zahtjev” svi uređaji mogu tražiti pristup sabirnici, kako bi postali inicijatori prenosa. Ako je sabirnica slobodna, arbitar aktivira signal “DS” - dodjela sabirnice, koji prolazi kroz niz ulančanih uređaja koji ne traže sabirnicu do onoga koji je tražio sabirnicu. Taj uređaj prekida lanac arbitriranja i dobija sabirnicu i deaktivira signal “Zahtjev”. Na kraju ciklusa prenosa na sabirnici, uređaj koji je postao inicijator aktiviranjem signala “Puštanje” obavještava sve ostale uređaje i arbitra da oslobađa sabirnicu i prepušta sljedećem arbitriranju. Tada arbitar deaktivira “DS” signal a zadnji vlasnik sabirnice deaktivira signal “Puštanje”. Na taj način, ne može se desiti da dva uređaja dobiju sabirnicu u istom arbitriranju. **Arbitar sabirnice** ne zna koliko uređaja zahtjeva sabirnicu, već samo ima li ili nema zahtjeva. **Prioritet** je definisan položajem uređaja u lancu, u odnosu na arbitar - najbliži (Uređaj 1) ima najviši prioritet. Da se nebi desilo da uređaj višeg prioriteta preotme sabirnicu u toku prenosa uređaja nižeg prioriteta, svi uređaji moraju voditi računa, ne samo o nivou “DS” signala na svom ulazu, već o njegovoj uzlaznoj ivici u sekvenci arbitriranja. Samo onaj koji traži sabirnicu i dobije uzlaznu ivicu signala dodjele, može zauzeti sabirnicu. Isto tako, bez signala “Puštanje”, arbitar na sabirnici nebi imao jasnu sliku o tome kada je završeno korištenje sabirnice, jer drugi zahtjevi za sabirnicu mogu stalno biti aktivni.
Ovakav način arbirtriranja je vrlo jednostavan za realizaciju i nije skup. Brzina mu je ograničena zbog ulančavanja signala dodjele, što unosi kašnjenja koja se sabiraju u dužim lancima. O maksimalnom kašnjenju arbitar mora voditi računa prilikom nadgledanja zauzimanja sabirnice. Zanemarivanje (izgladnjivanje, eng. - starvation) uređaja nižeg prioriteta je teško izbjeći. Jedno od rješenja ovog problema je uvođenje pravila da uređaj koji je upravo završio korištenje sabirnice, ne može je ponovo tražiti sve dok se signal zahtjeva ne deaktivira (nema drugih zahtjeva na čekanju).

Centralizovano paralelno arbitriranje - višenivoski model zaobilazi osnovni princip dodjeljivanja prioriteta zasnovan na udaljenosti od arbitra, time što se uvodi više nivoa (ili lanaca) arbitriranja. Višenivoski model bez ulančavanja uređaja je dat na slici 11.13.



Slika 11.13. Centralizovano paralelno arbitriranje bez ulančavanja uređaja. Primjer - PCI sabirnica

U ovom slučaju, svaki uređaj ima odvojene linije za zahtjevanje i dodjelu sabirnice. Arbitar je složenije strukture nego u prethodnom slučaju, jer mora nadgledati sve zahtjeve istovremeno i odlučivati kome dodjeliti sabirnicu aktiviranjem njegovog signala dodjele. Prioriteti mogu biti fiksni – ožičeni u arbitru, ili mogu biti promjenjivi promjenom pravila arbitriranja u arbitru. Dinamičko mijenjanje prioriteta može poboljšati raspodjelu sabrnice među uređajima, bez obzira na njihovu udaljenost od arbitra – nema ulančavanja uređaja.
Centralizovano paralelno arbitriranje sa ulančavanjem je prikazano na slici 11.14.

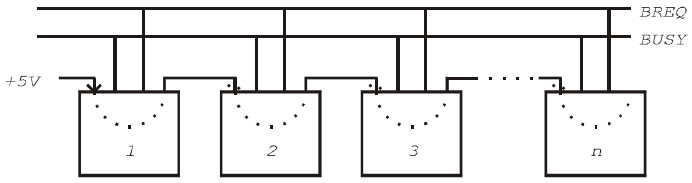


Slika 11.14. Centralizovano arbitriranje - višenivoski model, sa ulančavanjem uređaja. Primjer – VME sabirnica.

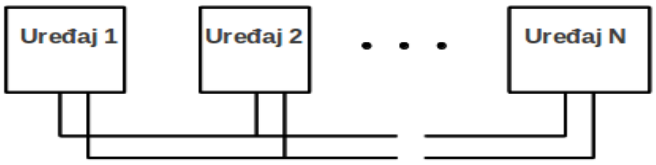
Svaki urađaj se veže na jedan od nivoa arbitriranja. Na predhodnoj slici uređaji 1 i 2 su vezani na nivo 1, uređaj 3 na nivo 2, sve do uređaja *n* na nivou *N*. Uređaji sa visokim prioritetom opsluživanja se vezuju na više nivoe arbitriranja (nivo *N* - najviši). Ako se jave zahtjevi na više nivoa prioriteta, sabirnica se dodjeljuje najvišem nivou. Među uređajima vezanim na isti nivo arbitriranja, važi princip jednonivoskog arbitriranja, uređaj 1 ima viši prioritet od uređaja 2 iako su oba na *lancu arbitriranja 1*. Uređaj 2 ima najniži prioritet u sistemu jer je na kraju lanca najnižeg prioriteta. Iako nije neophodno ožičavati veze svih modula sa svim nivoima arbitriranja, to se u praksi pokazuje pogodnim zbog omogućavanja naknadnih modifikacija. Ovaj metod zahtjeva dodatne linije na sabirnici i dodatnu logiku na ulazno/izlaznim uređajima, ali se na ovaj način bolje iskorištava sabirnica.

Distribuirano (decentralizovano) arbitriranje predstavlja model arbitriranja kod kojega je uloga arbitra distribuirana po svim uređajima na sabirnici koji na nju pretenduju. U nekim računarskim sistemima se i procesori "takmiče" za dobijanje sabirnice. Pri tome često imaju najniži nivo prioriteta u sistemu, jer većina ulazno/izlaznih uređaja može da izazove zastoj u

protoku podataka ukoliko dugo čeka na sabirnicu (npr kontroleri diskova, mrežni kontroleri, kontroleri terminala itd.).
Primjer ovakvog arbitriranja se nalazi u VAX arhitekturi firme Digital Equipment. Na VAX SBI sabirnici nema arbitra, već postoji 16 linija za traženje sabirnice. Svakom od maksimalno 16 modula, koliko može stati na stražnju sabirnicu dodjeljuje se po jedna. Svaki modul zahtjeva sabirnicu preko svoje linije, a motri na sve ostale, pa na kraju svakog ciklusa svaki modul zna da li je on sljedeći na redu. Ovako je potrebno više linija na sabirnici ali nema arbitra. Time je pouzdanost mehanizma arbitriranja na sabirnici sistemski poboljšana.
MULTIBUS® sabirnica, firme Intel, koristi 3 linije bez obzira koliko je uređaja na sabirnici. Kada niko ne traži sabirnicu, linija za arbitriranje propagira kroz sve uređaje. Da bi uređaj tražio sabirnicu prvo ona mora biti slobodna i njegov ulaz linije za arbitriranje mora biti aktivan. Tada uređaj prvo prekida lanac arbitriranja i svim uređajima iza sebe onemogućava traženje sabirnice. Samo jedan uređaj može imati "1" na svom ulazu i "0" na svom izlazu lance arbitriranja. On postaje *inicijator*, postavlja *BUSY* signal i svoj izlaz u "1" i počinje prenos na sabirnici. Uređaj 1 ima najviši prioritet - ako više uređaja traži sabirnicu, onaj koji je najbliži početku lanca arbitriranja ima prioritet. MULTIBUS nudi, kao alternativno i centralno arbitriranje.



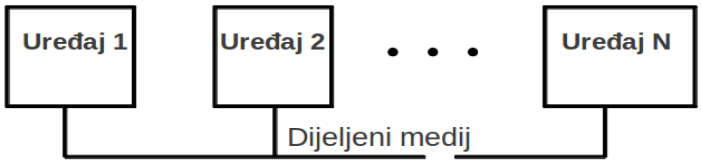
Slika 11.15 Arbitriranje na MULTIBUS sabirnici



Slika 11.16. Distribuirano arbitriranje - linije prenose zahtjeve za sabirnicom i identifikacione kodove uređaja i njihovih prioriteta. Primjer – NuBus u Apple Macintosh-u.

Posljednji primjer distribuiranog arbitriranja, je prikazan na slici 11.16. i predstavlja koncept po kome se vrši arbitriranje na NuBus-u, U/I sabirnici Apple Macintosh računara.
Svaki uređaj koji traži sabirnicu, objavljuje svoj zahtjev i identitet – nivo prioriteta. Na taj način, u svakom trenutku svi uređaji vide ko traži sabirnicu i koji mu je prioritet. Kako svaki uređaj zna svoj prioritet, u svakom trenutku može znati da li sabirnica pripada njemu, ili mora da sačeka da njegov zahtjev bude najvišeg prioriteta. Kao i u ranijim slučajevima arbitriranja sa fiksnim prioritetima, izgladnjivanje uređaja niskog prioriteta je moguće.

Arbitriranje otkrivanjem kolizije (eng. collision detection) na sabirnici (slika 11.17.) se primjenjuje u situacijama kada više uređaja dijeli jedan medijum – tipični primjer je Ethernet.



Slika 11.17. Arbitriranje otkrivanjem kolizije na sabirnici – dijeljenom mediju (npr. Ethernet)

Svaki uređaj može osluškivanjem ustanoviti da li je sabirnica slobodna. Ako jeste, zauzima je početkom prenosa podataka. Ako nije, nastavlja sa osluškivanjem dok ona ne postane slobodna. Ako se desi da više uređaja pokuša da zauzme sabirnicu istovremeno, svaki mora osluškivanjem ustanoviti da li je samo njegov saobraćajaj na sabirnici ili je došlo do kolizije sa još jednim ili više uređaja. Ako nema kolizije, nastavlja se prenos i sabirnica se smatra zauzetom. Ako je otkrivena kolizija, uređaj je dužan prekinuti prenos – osloboditi sabirnicu i pokušati ponovo nakon nekog vremena zadrške. Kako bi se smanjila mogućnost da uređaji koji su došli u koliziju ne dođu ponovo u konflikt nakon nekog vremena , to vrijeme svaki uređaj odabire slučano. Ako se konflikt ipak ponovo desi, proces se ponavlja. Ovo je vrlo jedostavan mehanizam arbitriranja, ali je očigledno da se vrijeme gubi na višestrukim kolizijama. U slučajevima kada saobraćaj na sabirnici nije gust i u njemu ne učestvuje veći broj uređaja, ova metoda se pokazuje kao vrlo efikasna.

11.1. Upravljanje U/I uređajima

Programi (programeri) upravljaju U/I uređajima u savremenim računarima na visokom nivou apstrakcije. Tipične ulazno-izlazne operacije u jezicima visikog nivoa (npr. *printf* i *scanf* u C-u) ne vode računa o formatu podataka koji se prenose, kodiranju podataka u formu u kojoj su razumljivi U/I uređaju, hardverskim povezivanjem sa uređajem i odgovarajućom vremenskom sekvencom događaja. O tim problemima na nižem nivou apstrakcije računa vodi sistemski softver. O pretvaranju jednog formata podataka u drugi, najčešće brinu U/I biblioteke programa-prevodioca (eng. compiler) koje se linkuju u objektni kod. O detaljima upravljanja U/I uređajima vode računa posebni upravljački programi (eng. device drivers) koji time rasterećuju korisnički program i omogućuju mu rad na višem nivou apstrakcije – ugodnije upravljanje. Obradu prekida ili izuzetaka rade posebni dijelovi jezgre (eng. kernel) operativnog sistema (OS). Upravljanje dijeljenjem U/I uređaja među više korisnika i/ili programa, takođe radi operativni sistem. Preslikavanje logičkih datoteka (eng. files) u fizičke uređaje i obezbjeđivanje kontrolisanog pristupa korisnicima, omogućava dio OS-a koji se zove datotečni sistem (eng. file system). Sve ovo omogućava programeru da apstrahuje nekoliko nivoa složenosti U/I sistema i koncentriše se na rješavanje svojih zadataka.
Kao što se vidi iz tabele 11.2. različiti U/I uređaji generišu različite količine podataka, pa upravljanje njima predstavlja različito opterećenje za procesor. Tastatura ili miš generišu do nekoliko desetina bajta u sekundi, a procesor mora odreagovati na svaki primljeni karakter sa ovih uređaja – ne može se tolerisati da pritisak na neko dugme na tastaturi bude propušten događaj. Sličan zahtjev predstavlja dolazak paketa podataka u mrežni komunikacioni kontroler, ali je ovdje brzina komunikacije i do nekoliko stotina MB/s. Propuštanje da se pročitaju pristigli podaci bi uzrokovalo njihovu retransmisiju i zastoj u komunikaciji, što se ne smije tolerisati. Sa druge strane, osvježavanje ekrana sadržajem video-memorije se mora odvijati najmanje 25 puta (slika) u sekundi. Ovo se može raditi automatski, unaprijed zadati

broj puta u sekundi, ali ako se radi o video zapisu visoke rezolucije (npr. 1920x1200=2304000 tačaka) i standardne dubine boja (24 bita) uz učestanost osvježavanja od 60Hz, procesor mora obraditi i smjestiti u video-bafer u memoriji oko 400MB podataka o slici svake sekunde.
Za svaku od ovakvih U/I operacija procesor mora komunicirati sa kontrolerima U/I uređaja. Da bi komunicirao, on mora vidjeti njihove vezne registre (najčešće kontrolne, statusne i registre podataka) u svom adresnom prostoru. Ovi registri se često nazivaju vratima (eng. ports), jer kroz njih procesor pristupa vanjskom svijetu. Kroz kontrolne registre procesor izdaje komande uređaju i proslijeđuje sve potrebne parametre – nrp. naredbu za pozicioniranje glava diska na određeni sektor. Kroz statusne registre procesor očitava trenutno stanje uređaja – npr. da li su podaci sa diska spremni. Registri podataka su vrata kroz koja dolaze ili odlaze podaci prema uređaju. Sva komunikacija sa U/I uređajem se odvija kroz njegov skup vrata-registara. Svi pomenuti registri moraju imati svoje adrese, kako bi procesor (program) mogao da ih proziva radi čitanja ili pisanja.

11.2. Adresiranje U/I uređaja

Postoje dva osnovna načina adresiranja U/I uređaja.
Njihove adrese mogu biti dio memorijskog prostora (eng. **memory mapped I/O**), odnosno dio memorijskog prostora se odvaja za U/I uređaje. Tako se gubi (mali) dio memorijskog adresnog prostora, a procesor pristupa U/I uređajima i njihovim veznim registrima istim instrukcijama (ali različitim adresama!) kao i svim ostalim memorijskim lokacijama. Naravno, fizička memorija ignoriše pristupe dijelu memorijskog prostora odvojenog za U/I uređaje.
Drugi način adresiranja je da se koristi odvojeni U/I adresni prostor (eng. **U/I mapped I/O**), mnogo manji od memorijskog (dovoljno je 10 do 15-bitni adresni prostor). Razlika u pristupu U/I uređajima u odnosu na memorijski prostor sada nije samo u odabranim adresama već i u različitim upravljačkim signalima. Isti opseg adresa se može koristiti u oba adresna prostora. Kod odvojenog U/I adresnog prostora, kariste se posebne instrukcije za pristup veznim registrima kontrolera uređaja – npr. umjesto *load* i *store* za pristup memorijskim lokacijama, koriste se *in* i *out* instrukcije.

11.3. Programsko prozivanje i prekidi

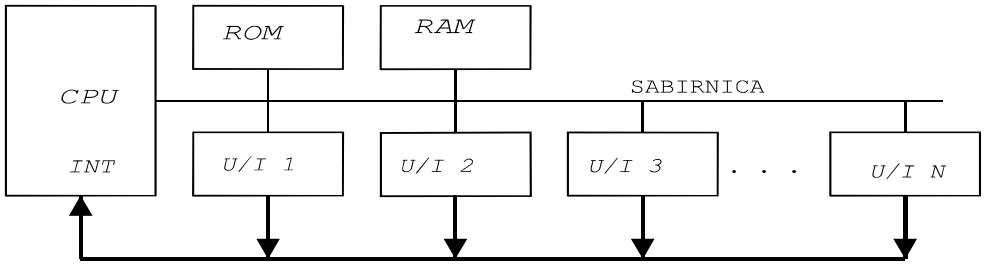
Prije prenosa podataka prema i od U/I uređaja, program(er) mora prvo provjeriti status tih uređaja kako bi ustanovio da li su uređaji spremni. Najjednostavnija metoda provjere statusa uređaja je **prozivanjem** (eng. *polling*) svakog od njih – jednog po jednog. Procesor testira statusni bit kontrolnog registra svakog od uređaja i na taj način “pita” uređaj da li je spreman - treba li mu usluga. Imajući na umu da miš ili tastatura moraju biti uslužen najmanje 30 puta u sekundi (da se nebi izgubio nijedan pokret miša ili pritisak na tastaturu), procesor već ima puno posla sa najsporijim perifernim uređajem. Iako bi magnetni disk sa brzinom prenosa od 80MB/sek prenosio podatke u blokovima od 16 bajta, bilo bi ga potrebno usluživati više od 5000000 puta u sekundi (nijedan blok se ne smije propustiti!). Lako je zaključiti da bi u ovom slučaju procesor gubio mnogo dragocjenog vremena čekajući da neki od uređaja bude spreman za prijenos podataka - zatraži njegove usluge.

Mnogo bolja metoda upravljanja U/I uređajima bi bila ona koja bi omogućila procesoru da više radi druge poslove, a prekida ih da usluži uređaj samo kada je ovaj spreman – traži

uslugu. Da bi to bilo moguće, procesoru se mora dovesti vanjski asinhroni ulaz za prekid (eng. interrupt), koji bi mu signalizirao da, kada završi instrukciju koju trenuto izvršava, skoči u program za usluživanje uređaja kome je usluga potrebna. Nakon usluživanja, procesor bi se vratio u glavni program – tamo gdje je stao. Lako je uočiti da je, uvođenjem opisanog mehanizma, efikasnost računarskog sistema značajno povećana. Ovaj mehanizam se zove **mehanizam prekida**.

Najjednostavniji mehanizam prekida je jednonivoski bez prioriteta – slika 11.18. Svi uređaji preko iste linije prekida mogu tražiti uslugu od procesora. Procesor prihvata prekid tako što, nakon završetka instrukcije koju izvršava, skače u rutinu za obradu prekida u kojoj:

- zabranjuje prihvatanje narednog prekida (“maskira” prekide)
- pronalazi uređaj koji je tražio prekid,
- pruža uslugu tom uređaju,
- daje nalog uređaju da deaktivira zahtjev za prekidom i
- omogućava prihvatanje narednog prekida (“demaskira” prekide).

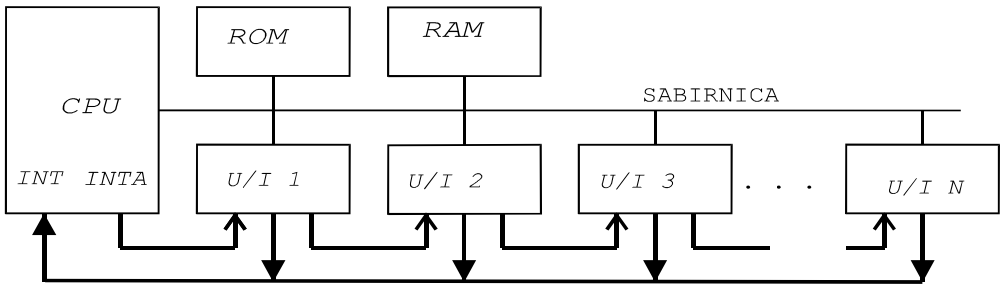


Slika 11.18. Jednonivoski sistem prekida bez prioriteta

Svi uređaji imaju isti prioritet prilikom traženja prekida, ali se uslužuju redom određenim u programu za obrade prekidâ. Ako više od jednog uređaja traži prekid, usluživaće se jedan po jedan, kroz više uzastopnih prihvatanja prekida.

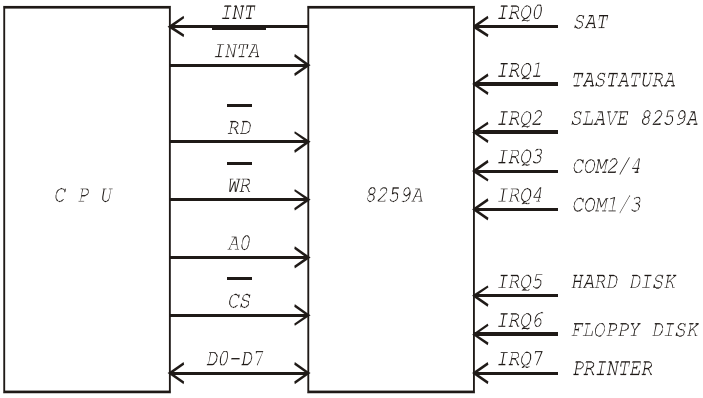
Nedostatak ovakvog sistema prekida je nepostojanje sistema prioriteta, pa se može desiti da se neki važan događaj ne bude obrađen na vrijeme, jer procesor nije stigao odgovoriti na njegov prekid.

Nešto složeniji sistem prekida koji uključuje jednonivosko arbitriranje po prioritetima, zahtjeva od procesora da svojim izlaznim signalom za prihvatanje prekida - INTA (od eng. *Interrutp Acknowledge*) odredi trenutak kada prihvata prekid – slika 11.19. Mehanizam arbitriranja je isti kao onaj kod arbitriranja za dodjelu sabirnice. U lancu uređaja može postojati samo jedan koji je tražio i dobio prekid (on ne prosljeđuje INTA signal dalje). Taj uređaj je imao viši nivo prioriteta od svih koji su (eventualno) istovremeno tražili prekid. U principu, brži uređaji treba da imaju viši prioritet da bi se izbjegli zastoji u komunikaciji sa njima.



Slika 11.19. Jadnonivoski sistem prekida sa arbitriranjem

Da bi se procesoru olakšao posao prozivke i pretraživanja ko je tražio prekid, uveden je mehanizam identifikacije uređaja koji je dobio pravo prekida. U trenutku dobijanja signala INTA, taj uređaj na sabirnicu podataka postavlja svoj jedinstveni identifikacioni broj – **vektor**. Procesor u **ciklusu prihvatanja prekida** čita taj vektor, i na osnovu njega računa i skače na adresu početka rutine za obradu toga prekida. Time je usluživanje prekida postalo jednostavnije i brže.
Višenivoski sistem prekida zahtijeva složeniji mehanizam arbitriranja. U mikroprocesorski baziranim računarskim sistemima problem realizacije takvog mehanizma prekida se obično rješava dodjelom prioriteta uređajima i korištenjem centralizovane šeme arbitriranja.
Za tu svrhu postoje programabilni kontroleri prekida, kao što je onaj u IBM-PC računarima, sa oznakom I8259A. Njegova veza sa procesorom, s lijeve strane i perifernim uređajima, s desne, je data na sljedećoj slici 11.20.



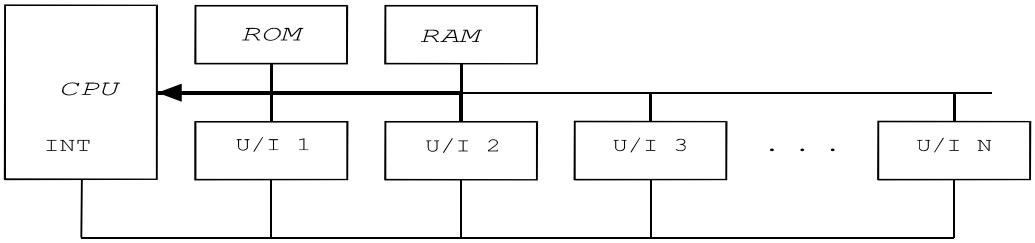
Slika 11.20. Upravljanje prekidima kod IBM-PC AT računara

Ovakvi kontroleri predstavljaju glavnu upravljačku strukturu u prekidima-upravljanim računarskim sistemima. Oni prihvataju zahtjeve od uređaja, određuju koji je od njih najvažniji (najprioritetniji), utvrđuju da li ima viši prioritet od onoga koji se, eventualno, trenutno uslužuje, i na osnovu toga odlučuje da li treba generisati novi zahtjev za prekidom procesora. Za svaki uređaj obično postoji poseban program – servisna rutina za obradu prekida, koji odgovara specifičnim potrebama tog uređaja. Kontroler zato treba, nakon traženja prekida, "vektorisati" procesor na rutinu koja obrađuje traženi prekid.
Kada jedan od 8 ulazno/izlaznih uređaja, vezanih za kontroler prekida po nivoima prioriteta, želi da izazove prekid procesora, postavlja svoj signal zahtjeva prekida (*IRQ*, od eng. *Interrupt ReQuest*, *IRQ0* ima najviši prioritet). Kada jedan ili više uređaja traži prekid, tada

kontroler aktivira *INT* ulazni signal CPU-a, na šta ovaj, kada završi instrukciju koju trenutno izvršava i ako prekid nije zabranjen (maskiran), odgovara prihvatanjem prekida *INTA* signalom. Tada kontroler "javi" procesoru koji od uređaja je tražio prekid, ispisivanjem njegovog vektora na sabirnicu podataka tokom ciklusa prihvatanja prekida. Taj podatak služi kao indeks u tabeli pokazivača (vektora) na programe za obrade prekida. 8259A sadrži više internih registara kojima CPU pristupa pomoću signala *RD* (čitanja), *WR* (pisanja), *CS* (Chip Select – linije za prozivanje kontrolera) i *A0* (za adresiranje registara kontrolera). Kada se obradi prethodni zahtjev za prekidom, CPU upisuje poseban kod u jedan od registara kontrolera, da bi 8259A deaktivirao INT liniju (osim ako ima još zahtjeva za prekidima). Pomoću tih registara se kontroler prekida može konfigurisati za rad na različite načine, maskirati neki od njegovih ulaza itd. Ako je potrebno vezati više od 8 I/O uređaja, vrši se kaskadno vezivanje kontrolera - maksimalno po još jedan kontroler na svaki od ulaza (8x8=64 ulaza), za šta postoje posebni signali.

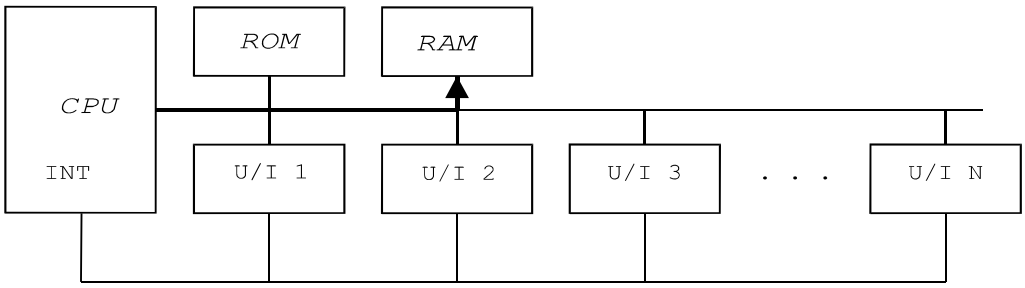
11.4. Direktni pristup memoriji

Računarski sistem podržan sistemom prekida radi mnogo efikasnije od onog kod kojeg procesor pretražuje ulazno-izlazne uređaje u potrazi za onim koji su spremni za prijenos podataka. Međutim, kada se javi zahtjev za prekidom, neki od uređaja, najčešće, traži da se u njega upiše blok podataka iz glavne memorije ili da se iz njega prenese blok podataka u glavnu memoriju (npr. disk kontroler, mrežni kontroler itd.).
Najjednostavniji način usluživanja ovakvih zahtjeva bi bio da procesor prenese čitav blok, čitanjem i pisanjem riječ po riječ. Neka je, kao na slici 11.21., ulazno-izlazni uređaj br. 2 tražio prekid radi prenosa bloka od N riječi u glavnu memoriju. Procesor će, u rutini za obradu zahtjeva, N puta odraditi ciklus čitanja iz U/I 2.



Slika 11.21. Procesor čita riječ iz U/I 2 uređaja

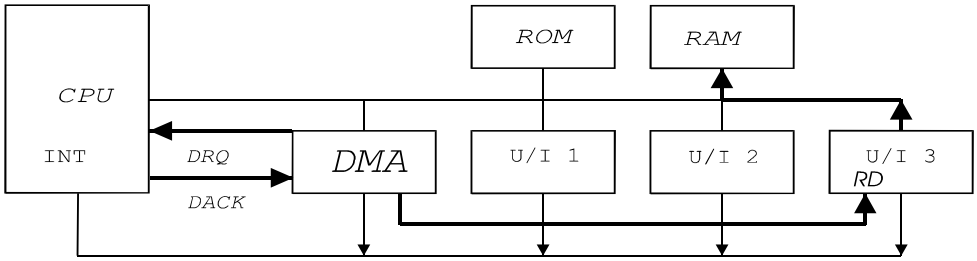
praćen ciklusom pisanja u memoriju, kao na slici 11.22.



Slika 11.22. Procesor piše pročitanu riječ iz U/I 2 u memoriju

Za prenos bloka od N riječi iz U/I 2 u RAM, procesoru treba program sa petljom od N

prolaza kojom obavlja 2N ciklusa prenosa na sabirnici. Zato procesor nije pogodan za ovakve poslove. Umjesto toga, ovakve poslove procesor može povjeriti posebnim upravljačkim strukturama za upravljanje prenosom na sabitnici – bez njegovog učešća, kontrolerima za direktan pristup memoriji (DMA, od eng. *Direct Memory Access*). DMA kontroler je još jedan od uređaja vezanih za sistemsku sabirnicu, koji ima mogoćnost da izazove prekid procesora. Kada procesor ustanovi da U/I uređaj traži da se prenese blok u memoriju, taj posao povjerava DMA kontroleru tako što mu, preko njegovih upravljačkih registara, kaže iz kojeg uređaja koliko riječi treba da prenese u glavnu memoriju i na koje mjesto. Od tog trenutka nadalje, DMA kontroler preko svojih signala za traženje (DRQ od eng. *DMA ReQuest*) i dobijanje (DACK od eng. *DMA ACKnowledge*) sabirnice, preuzima sabirnicu i koordinira prenos N riječi u N ciklusa prenosa na sabirnici.



Slika 11.23. Prenos iz U/I 3 u memoriju direktnim pristupom memoriji, upravljan DMA kontrolerom

DMA kontroler preko RD signala U/I uređaja upravlja ispisom riječi na sabirnicu, generiše odredišne adrese u memoriji i odrađuje memorijski ciklus pisanja. U/I uređaj vodi računa da nakon svakog prenosa, sljedeća riječ bude spremna (npr. FIFO baferom u kontroleru uređaja). Na kraju prenosa zadnje riječi, DMA kontroler izaziva prekid procesora, kako bi ga obavijestio o završetku prenosa. Ovim se procesor rasterećuje većine "rutinskog" posla u komunikaciji između U/I uređaja i glavne memorije, a i posao se brže završi.

Još naprednije računarske arhitetkure koriste **U/I procesore**, strukture koje su inteligentnije od DMA kontrolera. Kao što se DMA kontroler može programirati da uradi prijenos jednog bloka podataka, U/I procesor se može programirati da uradi sekvencu operacija sličnih jednoj DMA operaciji – jedan prenos bloka sa diska u memoriju, jedan blok drugačije veličine iz memorije ka mrežnom kontroleru i blok treće veličine iz memorije prema štampaču. Svaki od ovih poslova se posebno opisuje U/I procesoru kroz instrukcije za I/O operacije. Ovakve instrukcije se mogu izvršavati iz glavne memorije ili iz privatne memorije U/I procesora. Time je glavni procesor dodatno oslobođen operacija upravljanja saobraćajem sa U/I uređajima.

Direktni pristup memoriji predstavlja direktnu komunikaciju (bez učešća glavnog procesora) između memorije i jednog ili više U/I uređaja. Problemi koji se pri tome mogu javiti vezani su za činjenicu da se pod pojmom memorije krije čitava memorijska hijerarhija. Naročito probleme izazivaju keševi i virtuelna memorija. Kada procesor pristupa glavnoj memoriji, on je vidi kao virtuelnu memoriju sa virtuelnim adresama. Sa druge strane, DMA kontroler radi sa fizičkim adresama. Prema tome, kada procesor inicijalizira DMA kontroler, on mu mora prevesti virtuelnu u fizičku adresu početka bloka u memoriji. Problem je u tome što se virtuelni adresni prostor ne mora prevoditi (preslikavati) linearno u fizički, susjedne stranice u virtuelnoj memoriji ne moraju biti preslikane u susjedne stranice u fizičkoj. Zato, u slučaju

prenosa većih količina podataka od jedne stranice, nije dovoljno prevesti jednu početnu adresu bloka u memoriji. Rješenje može biti da se podaci podijele na blokove veličine do jedne stranice, ali se na taj način povećava učešće procesora u prenosu podatakak kroz povećani broj inicijalizacija DMA kontrolera. Kako savremeni procesori direktno komuniciraju sa kešom (kontrolerom), a DMA bazirani prenosi nemaju pristup kešu, nastaje problem konzistentnosti podataka u memorijskoj hijerarhiji. Nakon DMA prenosa, u memoriji se može naći podatak svježiji od onoga u kešu. Može se desiti i obrnuto, da procesor ažurira podatke u kešu, a odgađa pisanje bloka novih podataka i u glavnu memoriju – svaki DMA prijenos takvog bloka iz memorije ka U/I uređaju će izazvati slanje zastarjelih podataka. Zato je neophodno korištenje nekog niza pravila za održavanje koherentnosti keša (eng. Cache coherency protocol). Time se kontrolerima keša daje dodatni posao koji obezbjeđuje da se ne desi da neko u sistemu koristi zastarjele kopije podataka.

12. Paralelne računarske arhitekture

Iako osnovna frekvencija sata kod savremenih računarskih arhitektura neprestano raste, brzina logičkih kola se ne može povećavati u nedogled. Brzina svjetlosti (kretanja elektrona, protona i fotona) postaje glavni ograničavajući faktor projektantima superkompjutera. Odmah do njega je problem hlađenja sistema napravljenih od poluprovodničkih struktura najvišeg raspoloživog stepena integracije, međusobno povezanih najkraćim mogućim vezama. Pored toga, tranzistori, kao osnovni prekidački elementi, se tako gusto pakuju da će se uskoro broj atoma u kanalu svakog od njih moći lako prebrojati, a na tom nivou su zakoni kvantne fizike neumoljivi.
Zbog svega toga se sve više pažnje posvećuje paralelnim računarima. Ako nije moguće ubrzati savremeni procesor 1000 puta, moguće je pokušati zaposliti 1000 takvih procesora istovremeno u nadi da će se time postići ekvivalentne performanse.
U prethodnim poglavljima je bilo riječi o paralelizmu na nivou instrukcija, kaji se koristi u protočnim i superskalarnim strukturama za postizanje oko 10-strukog ubrzanja u odnosu na klasično sekvencijalno izvršavanje instrukcija. Daljnje višestruko ubrzanje nije moguće postići bez istovremenog korištenja velikog broja takvih procesora.

12.1. Projektovanje paralelenih računara

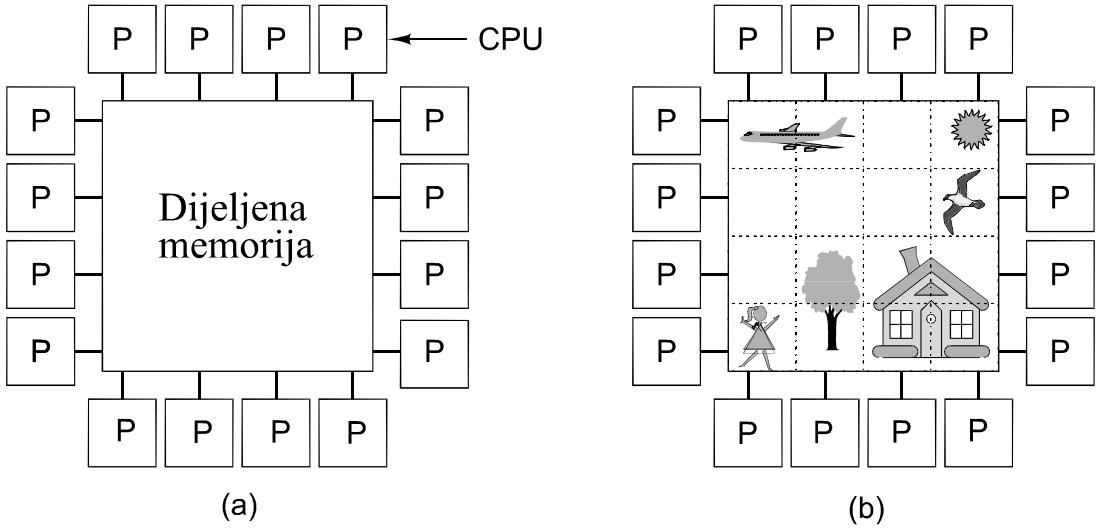
Prilikom projektovanja hardvera paralelnih računarskih sistema osnovni problemi su:

- koliko i kakvih procesirajućih elemenata odabrati,
- koliko i kakvih memorijskih elemenata odabrati i
- kako međusobno povezati izabrane elemente.

Procesirajući elementi su, najčešće, standardni mikroprocesori, iako to mogu biti i druge strukture, od aritmetičko-logičkih jedinica do kompletnih računara.
Memorijski sistemi se, obično, sastoje od modula različitih kapaciteta i brzina pristupa, hijerarhijski organizovanih sa dva, tri, ili više novoa keširanja.
Ono po čemu se paralelni računarski sistemi najviše razlikuju, je način na koji su ovi dijelovi povezani. Veze mogu biti statičke i dinamičke. Statičkim se nazivaju sva fiksna ožičenja bez obzira da li se radi o povezivanju u obliku zvijezde, prstena, mreže ili nekom drugom. Kod dinamičkog povezivanja, svi elementi se vežu na prekidačku spojnu mrežu, koja može mijenjati spojne puteve za razmjenu poruka između komponenti sistema.
Programska podrška paralelnim računarima, u najvećoj mjeri, zavisi od njihove namjene. Neki se projektuju za istovremeno izvršavanje više međusobno nezavisnih zadataka. Takvi programi i nemaju potrebu da međusobno često komuniciraju. S druge strane, moguće je izvršavati jedan zadatak sastavljen od više paralelnih procesa, kada procesori trebaju biti međusobno povezani brzim/visoko-propusnim vezama. U realnosti, različiti problemi koje treba rješavati pomoću paralelnih računara, a time i programi, algoritmi i strukture podataka koji se obrađuju, diktiraju projektovanje različitih arhitektura koje se mogu smjestiti između pomenutih dviju krajnosti.

12.2. Načini povezivanja komponenti sistema

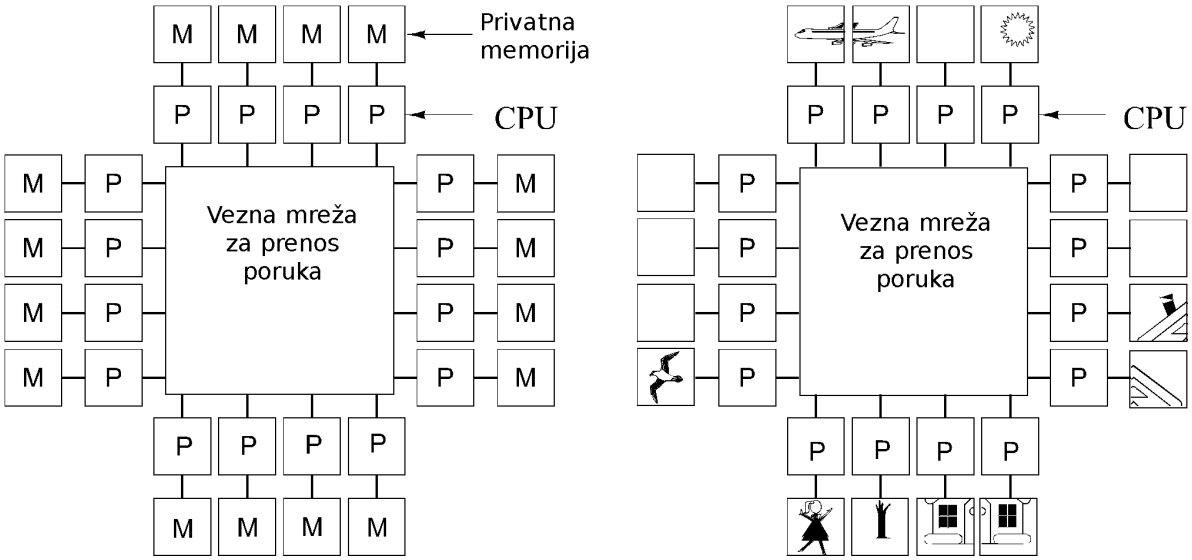
Na osnovu načina povezivanja komponenti sistema, postoje dva osnovna tipa paralelnih računarskih arhitektura – multiprocesorski i multiračunarski.
Kod multiprocesorskih, svi procesori dijele jednju zajedničku memoriju, kao na slici 12.1.



Slika 12.1 Multiprocesor sa 16 procesora i dijeljenom memorijom (a), slika za analizu, podijeljena na 16 dijelova, svakom procesoru po dio.

Drugo ime za njih je sistemi sa dijeljenom memorijom (eng. **Shared memory**). Ovakav pristup se reflektuje u programskom modelu na taj način da svi paralelni procesi dijele jedan virtuelni memorijski prostor preslikan u zajedničku memoriju. Sve što je potrebno da bi se obavljala komunikacija među procesima je čitanje i pisanje po memoriji. Ova jednostavnost programskog modela čini multiprocesorske sisteme vrlo popularnim. Ujedno, oni se mogu primijeniti za obrade pri rješavanje velikog broja različitih problema. Slikovit je primjer programa za obradu slike (bit-mape), koji treba da prepozna sve objekte na njoj (slika 12.1.b). Na svakom od 16 procesora se izvršava isti proces ali nad 16 različitih dijelova slike. To što svaki proces vidi cijelu sliku olakšava obradu jer se neki objekti prostiru u više dijelova slike. Tako isti objekat može prepoznati više procesa pa je na kraju potrebno koordinirati nađene rezultate.

Drugi tip paralelnih računara je onaj kod koga svaki procesor ima svoj memorijski prostor, kojem drugi procesori ne mogu direktno pristupati. Takvi sistemi se zovu multiračunari ili sistemi sa distribuiranom memorijom (eng. **Distributed memory**). Za veze među procesorima je potrebno obezbijediti posebne komunikacione kanale. Odsustvo dijeljene memorije ima za posljedicu usložnjavanje programskog modela i načina programiranja. Slika 12.2. ilustruje koliko je složenija obrada slike iz prethodnog primjera pomoću multiračunara. Dva procesora koji rade na prepoznavanju istog objekta moraju koristiti druge načine komuniciranja da bi dobili tražene podatke. Prvo moraju znati od koga trebaju tražiti podatke, a onda i odgovarati na slične zahtjeve drugih (što kod multiprocesora nije slučaj). Pored toga, podjela zadataka i podataka za obradu, zahtijeva mnogo komuniciranja.



Slika 12.2 Multiračunar sa 16 procesora, svaki sa svojom memorijom (a), slika za analizu podijeljena u 16 memorija (b).

Može se zaključiti da je programiranje multiračunara mnogo složenije (i skuplje) od programiranja multiprocesora. S druge strane, projektovanje multiračunara je mnogo jednostavnije (i jeftinije) od projektovanja multiprocesora sa istim brojem procesora. Projektovati računar sa dijeljenom memorijom sa šesnaest procesora je pravi poduhvat, a multiračunar sa istim brojem procesora procesora je relativno jednostavno.

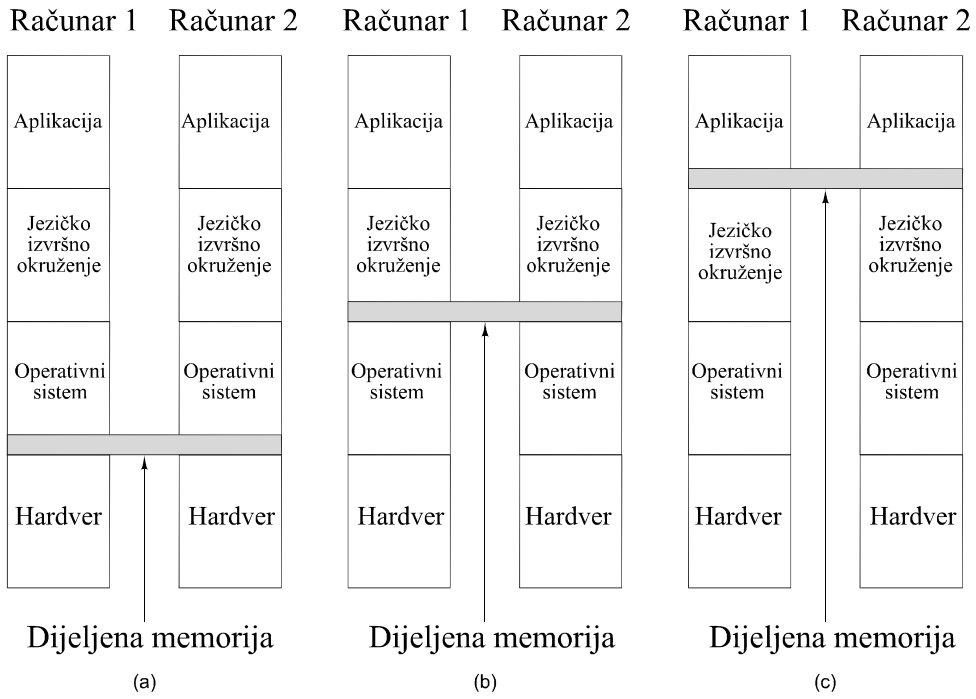
Ove dvije krajnosti su učinile da se pojavljuje sve veći broj hibridnih sistema koji se relativno lako projektuju i programiraju. U tom pravcu ide i većina savremenih istraživanja sa ciljem da se naprave sistemi koji se lako proširuju – nadograđuju, i tako im se povećavaju ukupne performanse.

Ako se računarski sistem posmatra kao hijerarhija nivoa apstrakcije, onda se dijeljena memorija može realizovati na više različitih mjesta, kao na slici 12.3.

Na slici 12.3.(a) je dijeljena memorija realizovana na u hardveru kao kod multiprocesora. Tu postoji jedna kopija operativnog sistema i pratećih struktura podataka. Operativni sistem vidi jednu memoriju (uključujući i virtuelnu) i vodi računa o pravima pristupa njenim pojedinim dijelovima.

Drugi način je da se koristi multiračunarska arhitektura sa operativnim sistemom koji simulira zajedničku dijeljenu virtuelnu memoriju na nivou čitavog sistema (eng. DSM – Distributed Shared Memory), slika 12.3. (b). Kada neki procesor pokuša da čita ili piše po stranici virtuelne memorije koja nije kod njega, javlja se prekid/izuzetak operativnom sistemu, koji locira traženu stranicu i traži od procesora kod koga se ona nalazi da njem sadržaj pošalje preko vezne mreže. Poslije remapiranja, ponovo se pokreće instrukcija koja je izazvala izuzetak. Za korisnika ova remapiranja nisu vidljiva i on ima utisak da radi sa standardnom dijeljenom memorijom kao pod (a).

Treći način je da se dijeljena memorija realizuje na nivou korisničkog okruženja za podršku jednom od jezika visokog nivoa. Tada programski jezik daje privid dijeljene memorije, koja se zatim realizuje pomoću kompajlera i korisničkog okruženja slika 12.3. (c).



Slika 12.3 Nivoi na kojima se može realizovati dijeljena memorija

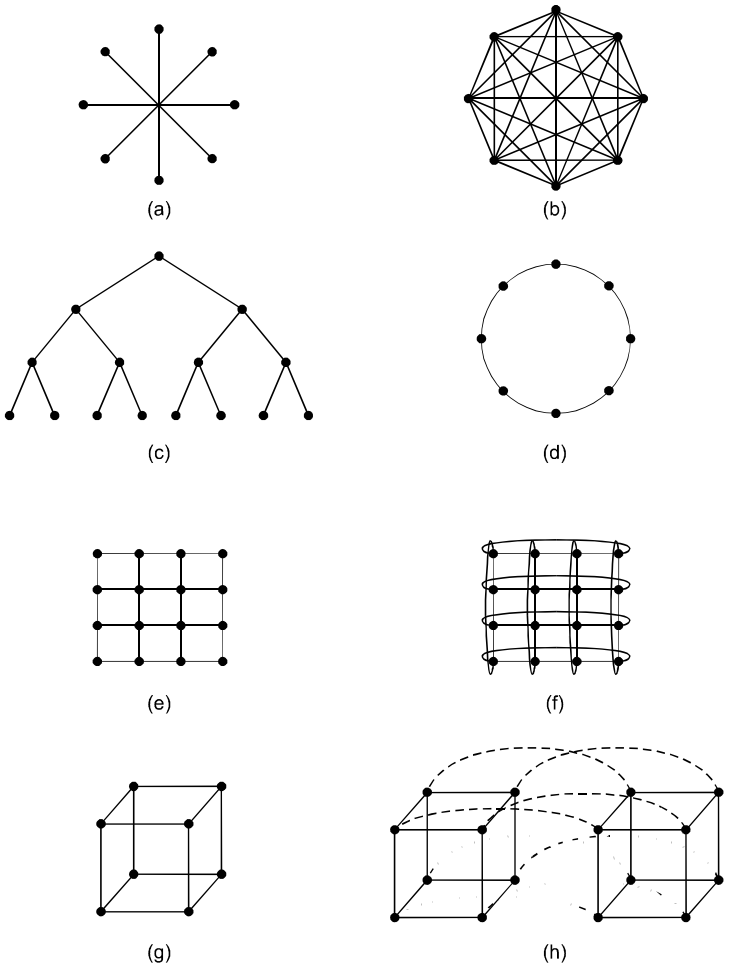
12.3. Mreže za povezivanje

Mreže za povezivanje, kod paralelnih računara, se mogu sastojati od:

1. - procesora,
2. - memorija,
3. - interfejsa,
4. - veza (linkova) i
5. - preklopnika (eng. switches).

Procesori i **memorije** su krajnje tačke, izvorišta i odredišta, komunikacija.
Interfejsi su uređaji koji donose i odnose poruke za procesore i memorije (najčešće vezni čipovi ili moduli smješteni uz procesore, koji mogu komunicirati sa procesorima i memorijama).
Veze su fizički kanali kroz koje se komunikacija ostvaruje. Mogu biti električni ili optički, serijski ili paralelni, jednosmjerni ili dvosmjerni. Karakteriše ih maksimalna propusnost, i definiše se kao broj bita koje mogu prenijeti za jednu sekundu.
Preklopnici su uređaji sa više ulaza i izlaza (portova). Kada poruka/paket stigne na neki ulaz, određeni biti iz nje/njega se koriste za određivanje preko kog izlaza će se slati dalje – proslijediti. Paketi su, obično, veličine od nekoliko bajta do nekoliko kilobajta.
Postoji analogija između spojnih mreža i ulica u gradu. Svaka ulica je kao veza – ima smjer, propusnost (brzinu) i širinu (broj linija). Raskrsnice su analogne preklopnicima – sa svakog ulaza, pješak ili vozilo ima na raspolaganju određen broj izlaza.
Topologija veznih mreža je određena rasporedom veza i preklopnika. Ona se može modelirati grafom sa vezama kao lukovima i preklopnicima kao čvorovima, kao na slici 12.4.
Svaki čvor/preklopnik ima određen broj veza kojima je povezan sa okolinom. Broj tih veza definiše **stepen** čvora (eng. degree, fanout). Što je viši stepen čvora, to je veća mogućnost

izbora povezivanja i neosjetljivost na grešku – pad neke veze.



Slika 12.4. Različite mrežne topologije. Prikazani su samo preklopnici i veze.

Diametar vezne mreže je definisan kao broj linkova između dva najudaljenija čvora – koji međusobno komuniciraju sa najvećim kašnjenjem. Kao parametar se spominje i srednja (prosješna) udaljenost između čvorova mreže.

Mnogi projektanti smatraju da je **bisekcijska propusnost** mreže najvažnija mjera njenog kvaliteta. Ona se izračunava tako što se mreža podijeli na dva jednaka dijela (prema broju čvorova), a zatim se sabere propusnost svih odstranjenih veza. Tom prilikom se uzima najniža propusnost od svih varijanti podjele mreže.

Dimenzionalnost mreže se definiše kao broj mogućih izbora puteva između dva čvora, izvorišta i odredišta. Ako nema niti jednog izbora spojnog puta (već samo jedan), mreža je 0-dimenzionalna.

Više topologija je dato na slici 12.4, sa vezama kao linijama a preklopnicima kao tačkama. Memorije i procesori se, obično, vežu na preklopnike pomoću interfejsâ.

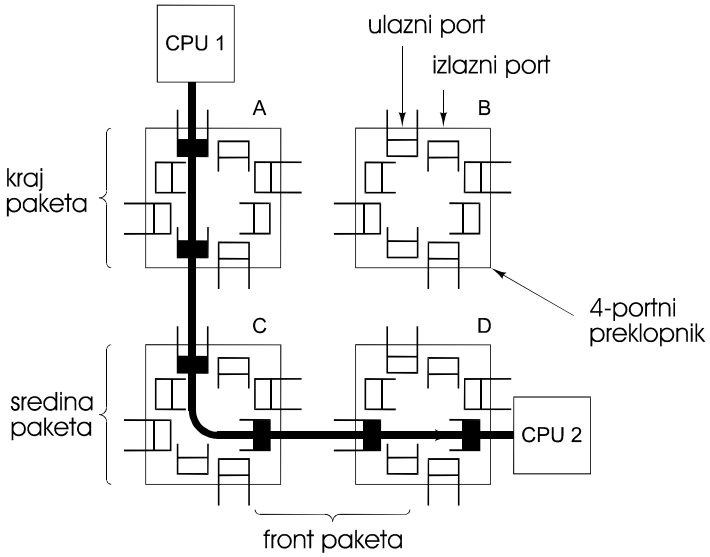
Pod (a) je data 0-dimenziona **zvijezda** konfiguracija sa momorijom i procesorima u vanjskim čvorovima i sa centralnim čvorom koji je samo preklopnik. Taj preklopnik je ujedno i usko grlo i slaba tačka sa stanovišta pouzdanosti/neosjetljivosti na greške.

Pod (b) je dat 0-dimenzioni dizajn ali sa punom povezanošću. Svaka dva čvora su direktno povezana, maksimalna je bisekcijska propusnost, minimalan dijametar i visoka neosjetljivost na greške. Kako je za k čvorova potrebno k(k-1)/2 linkova, za veliko k ovaj dizajn postaje

nepraktičan.
Pod (c) je data 0-dimenzionalna topologija – **stablo**. Kod nje je bisekcijska propusnost jednaka propusnosti jedne veze. Najveći saobraćaj se odvija oko vrha stabla, što je usko grlo dizajna. Topologija kod kojih se opterećenijim vezama/granama daje veća propusnost se naziva debelo stablo.
Prsten na slici (d) je jednodimenzionalna mreža (izbor lijevo ili desno).
Mreža/rešetka na slici (e) je dvodimenzionalni dizajn, vrlo često korišten u praksi. To je vrlo pravilna/regularna struktura, jednostavna za proširenje i ima dijametar koji se povećava sa korijenom broja čvorova.
Vrsta rešetke zvana **dupli torus** je data na slici (f). To je rešetka sa spojenim ivicama. Time se postigne veća neosjetljivost na greške i smanjuje se dijametar mreže – čvorovi iz suprotnih uglova komuniciraju kroz dvije veze.
Kocka na slici (g) je trodimenzionalna topologija, dok je na slici (h) data 4-dimenzionalna kocka satavljena od dvije 3-dimenzionalne, vezivanjem odgovarajućih vrhova. 5-dimenzionalna kocka bi se mogla napraviti povezivanjem, na isti način, dvije 4-dimenzionalne itd. Takva n-dimenzionalna kocka se zove **hiperkocka**. Dijametar ovakve strukture raste linearno sa dimenzionalnošću (jednak je logaritmu po bazi 2 od broja čvorova) i, iako je vrlo skupa za realizaciju, vrlo često se sreće u paralelnim arhtekturama. U poređenju sa rešetkom sa 1024 čvora (32 puta 32) koja ima dijametar 62, 10-dimenzionalna hiperkocka ima dijametar od samo 10, sto znači više od 6 puta kraća kašnjenja u komunikacijama.

12.4. Preklopnici

Vezne mreže se sastoje od preklopnika i ožičenja među njima. Slika 12.5. prikazuje malu veznu mrežu sa 4 preklopnika, svaki sa po 4 ulaza i 4 izlaza. Radom ovakvih uređajima obično upravlja ugrađeni procesor. Zadatak preklopnika je da prihvati dolazeći paket na ulaznom portu i proslijedi ga na određeni izlaz.



Slika 12.5. Vezna mreža sa 4 preklopnika - prenos paketa unaprijed zauzetom putanjom

Svaki izlaz je vezan na ulaz drugog preklopnika. Veze mogu biti serijske ili paralelne. Ako su serijske mogu biti sinhrone ili asinhrone, u oba slučaja relativno jednostavne za realizaciju i relativno spore. Ako su veze paralelne, zahtijevaju dodatne upravljačke signale i omogućavaju veće brzine prenosa podataka. Kod paralelnih veza je cijena realizacije

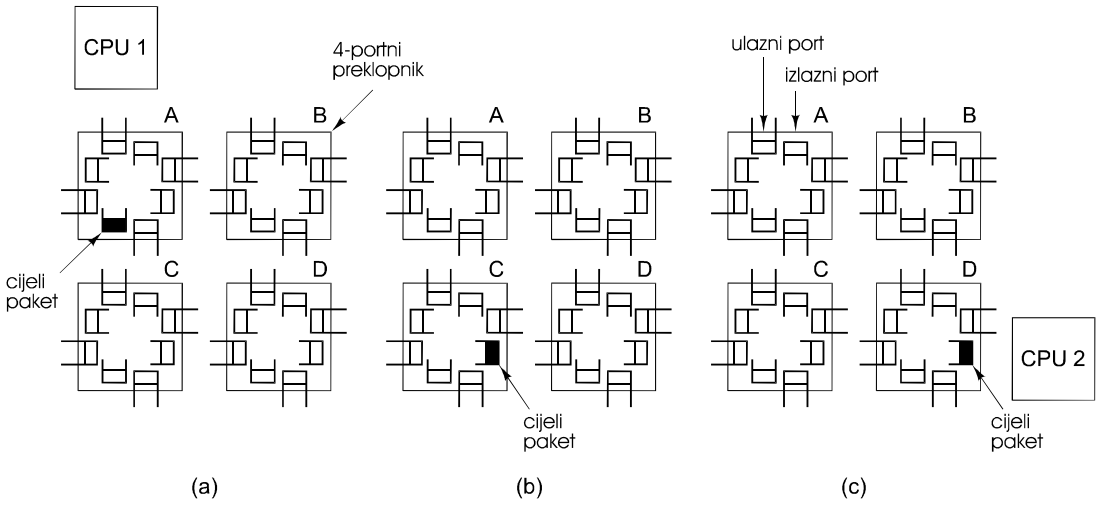
relativno visoka i javlja se problem međusobnog kašnjenje signala (eng. skew).
Preklopnici mogu raditi

1. komunikaciju kanala ili
2. komutaciju paketa.

U prvom slučaju se vezni put zauzme prije početka slanja podataka (slika 12.5). Kod domutacije paketa (eng. store and forward) manja je zauzetost resursa u odnosu na prvi slučaj.
Kod komutacije paketa postoje tri moguća tipa baferovanja podataka:

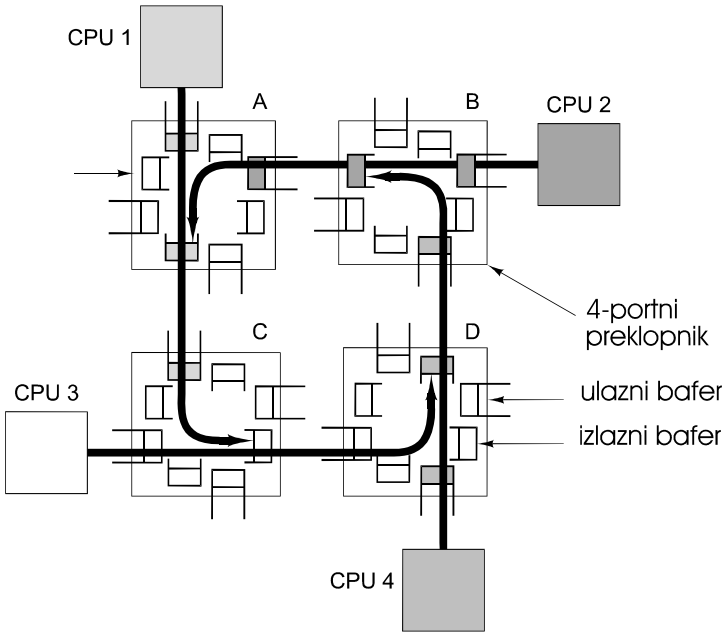
1. ulazno baferovanje (FIFO na ulazu), kada svi paketi iza prvog čekaju da on dobije slobodan izlaz, iako postoje paketi u redu čekanja čiji su izlazi slobodni (eng. head of line blocking),
2. izlazno baferovanje, kada se baferi nalaze na izlaznom portu pa poruke ne blokiraju one iza sebe, i
3. zajedničko baferovanje, kada su baferi zajednički i dinamički se alociraju - po potrebi (tada se problem administriranja usložnjava i može se desiti da zagušenje na jednom portu prouzrokuje smanjenje kapaciteta bafera za ostale).

Baferovanje ima svoj nedostatak - unosi kašnjenje. Na slici 12.6. vrši se 4 kopiranja podataka (u A, u C, u D i u CPU2).



Slika 12.6. Vezna mreža sa 4 preklopnika - prenos paketa sa baferovanjem

Kompromisno - hibridno rješenje između komutacije kanala i paketa bi bilo da se dijelovi paketa šalju komutiranim kanalima.
Preklopnici rade slično protočnim strukturama - "svaki po malo, a zajedno mnogo".
Pravilo koje određuje kuda će se rutirati paket u mreži dimenzionalnosti jedan ili više, naziva se algoritam rutiranja. Kada je ne raspolaganju više ruta, dobar algoritam će raspodjeliti saobraćaj na više puteva istovremeno, kako bi upotrijebio svu raspoloživu propusnost mreže. Pored toga, algoritam mora spriječiti "mrtve petlje" u spojnim mrežama (slika 12.7. za slučaj komutacije kanala) kada više paketa u prenosu zauzmu resurse tako da nijedan ne može proći dalje i tako ostanu zauvijek.

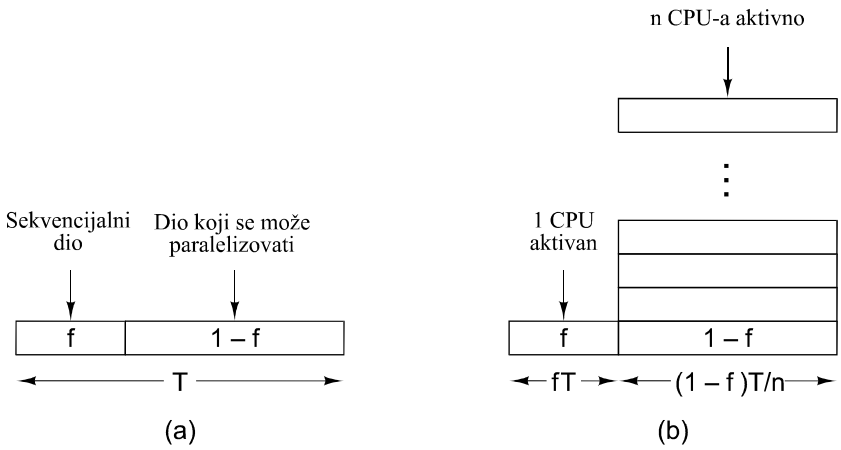


Slika 12.7. Mrtva petlja prilikom zauzimanja resursa u spojnim mrežama

Algoritmi rutiranja se mogu podijeliti na one kod kojih rutiranje određuje izvorište poruke i na one kod kojih je odlučivanje distribuirano. Distribuirani algoritmi se mogu podijeliti na statičke (paketi na isto odredište uvijek idu istim putem) i dinamičke (kada rutiranje zavisi od trenutnog stanja saobraćaja).

12.5. Performanse paralelnih računara

Sa stanovišta performansi CPU-a i U/I operacija, one se mogu mjeriti kao kod jednoprocesorskih sistema. Performanse međuveza se analiziraju kroz dva parametra - kašnjenje (eng. latency) i propusnost (eng. bandwidth).
Kašnjenje kod komutiranih linija je jednako zbiru vremena uspostave veze i vremena trajanja prenosa podataka. Kod komutiranjih paketa ono je jednako zbiru vremena uspostave paketâ i vremena njihovog prenosa.
Propusnost se može definisati kao ukupna (zbir kapaciteta svih veza) ili bisekcijska, o čemu je ranije bilo govora. Često se, kao mjera performansi, uzima i prosječna izlazna propusnost svakog CPU-a.
Komunikacija većim paketima povećava propusnost ali i kašnjenje. Komunikacija manjim paketima daje manju propusnost ali i manje kašnjenje. Ovaj konflikt se rješava u zavisnosti od važnosti jednog ili drugog parametra u konkretnim situacijama.
Propusnost se može povećati (kupiti), dodavanjem linija - proširivanjem sabirnica, ali kašnjenje ne.
Korisnike više interesuju "softverske" performanse - koliko će se brže izvršavati njihovi programi na paralelnim računarima u poređenju sa jednoprocesorskim sistemima. To, u najvećoj mjeri, zavisi od strukture programa koji se izvršavaju. Neka se neki program na jednoprocesorskom računaru izvršava **T** sekundi, od čega **f** označava dio vremena kada se izvršava sekvencijalni dio koda, a **(1-f)** dio koda pogodan za paralelno izvršavanje. Ako je moguće paralelni kod izvršavati na **n** procesora bez dodatnih zadataka koje bi to sobom nosilo, tada bi se vrijeme izvršenja smanjilo sa (1-f)T na (1-f)T/n. Tada bi ukupno vrijeme izvršenja bilo fT+(1-f)T/n.

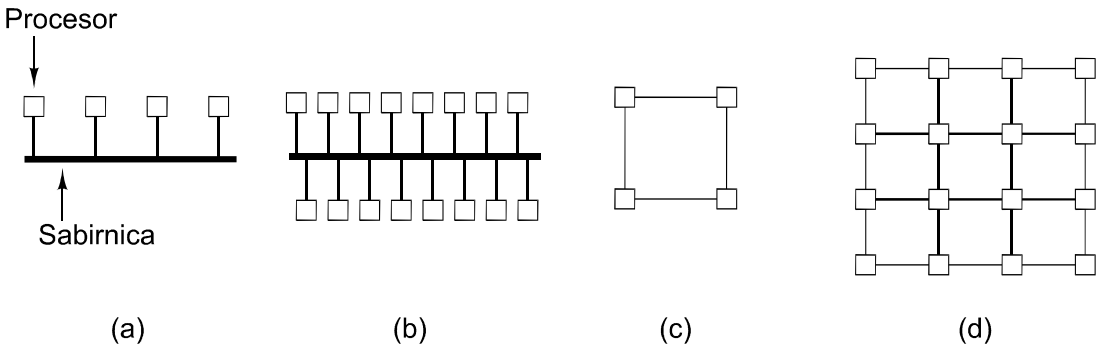


Slika 12.8. Moguće ubrzanje kod paralelnog izvršavanja programa

Ubrzanje se dobije kada se vrijeme izvršenja na jednoprocesorskom računaru podijeli sa onim na n-procesorskom:

$$ubrzanje = \frac{n}{1+(n-1)f}$$

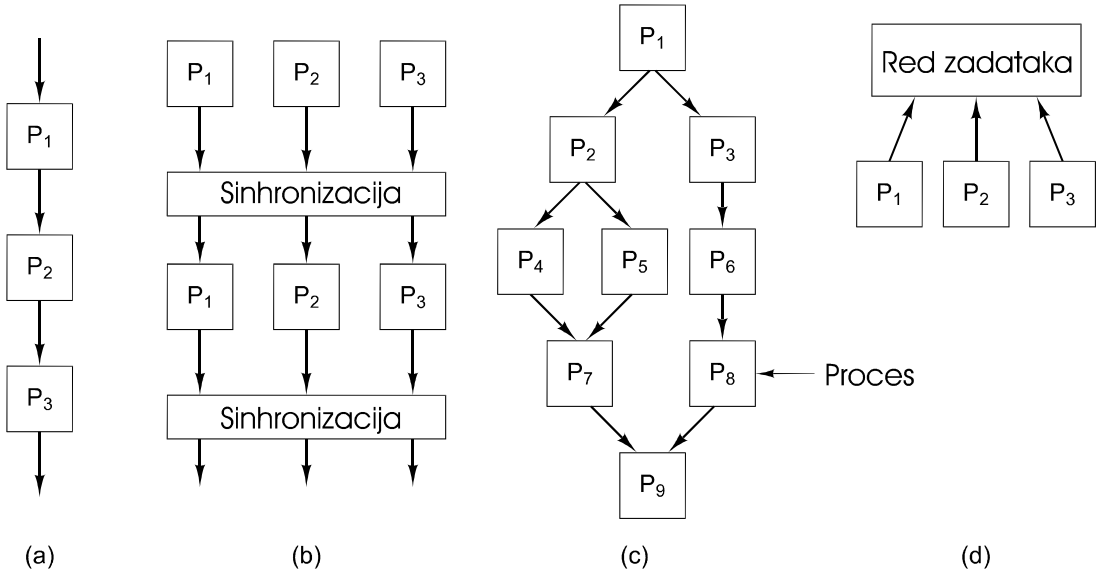
Ovaj izraz je poznat i kao Amdhal-ov zakon. U realnim uslovima, vrlo je teško postići i približno ubrzanje ovom "idealnom". Kašnjenja zbog komunikacija veznim putevima ograničene propusnosti, kao i nepogodnost algoritama izvršavanju na velikom broju procesora, su samo neki od razloga. Često se koriste i neoptimalni ali paralelni algoritmi. Zbog relativno niskih cijena hardvera, korištenje 2n procesora za ubrzanje od n puta se smatra efikasnim.
Ako se dodavanjem procesora postiže povećavanje efektivnih performansi sistema, takav sistem se naziva pogodnim za proširenje (eng. scalable).



Slika 12.9. Procesori povezani sabirnicama na nekoliko načina

Ilustracija analize pogodnosti za proširenje je data na slici 12.9. Ako se sistem sa 4 procesora na jednoj sabirnici (a) proširi na 16 procesora (b) raspoloživost sabirnice po procesoru pada sa 1/4 na 1/16 pa se ovakav sistem ne smatra pogodnim za proširenje. U rešetkasto povezanim sistemima dodavanje procesora podrazumijeva i dodavanje sabirnica. Odnos broja

sabirnica i procesora sa je 1 kod (c) i raste na 1,5 kod (d) - 16 procesra i 24 sabirnice. Dalje dodavanje povećava ukupnu raspoloživu propusnost po procesoru. S druge strane, dodavanje procesora na jednoj sabirnici ne povećava dijametar vezne mreže ili kašnjenja u prenosu poruka, kao kod rešetkastog povezivanja. Za rešetku n × n dijametar je 2(n-1) pa kašnjenje raste približno sa korijenom broja procesora (za 400 procesora dijametar je 38 a za 1600 procesora raste na 78).
Idealan sistem bi trebao održavati konstantnim propusnost po procesoru i kašnjenje, nakon povećanja broja procesora. Prvo je i u praksi ostvarivo, ali je nemoguće spriječiti povećanje kašnjenja (logaritamsko povećanje kašnjenja kod hiperkocke je jedno od najboljih rješenja). Kašnjenje najviše degradira performanse kod fino usitnjenih zadataka raspoređenih na veći broj procesora, kada se zahtijeva veći obim razmjene podataka.
Postoje posebne metoda "prikrivanja" kašnjenja. Jedna od takvih je **repliciranje podataka**, kada se više kopija podataka drži na više lokacija kako bi se smanjilo vrijeme pristupa. Odmah se javljaju pitanja oko toga gdje, kada i koji podaci će se kopirati, ko će tražiti kopiranje i ko će održavati međusobnu konzistentnost kopija. Odgovori leže u opsegu od hardverski zahtjevanog i određenog kopiranja u tranutku izvršenja programa, pa do kompajlerski predviđenog i orkestriranog repliciranja.
Druga metoda je **pred-dobavljanje** (dobavljanje podataka unaprijed). Keširanje je najčešći način, kada se na zahtjev programa donose traženi podaci ali i svi ostali iz bloka (linije keša) podataka iz memorije. Keširanjem može, indirektno, upravljati i kompajler kada zna da će programu uskoro trebati određeni podaci može "pogurati" te instrukcije unaprijed u programu kako bi se traženi podaci u međuvremenu dobavili.
Treća metoda je **višenitnost**, tehnika poznata u sistemima koji podržavaju izvršavanje više lakih procesa – niti (koji dijele isti memorijski prostor, pa promjena konteksta ne zahtjeva promjene u upravljanju virtuelnom memorijom) istovremeno (ili pseudo-istovremeno kroz vremensko dijeljenje procesora). Ako procesor može dovoljno brzo vršiti promjenu konteksa među više niti, tada se ne mora gubiti vrijeme na čekanje (npr. na dolazak udaljenih podataka koji trebaju procesu koji trenutno zauzima procesor), već se izvršenje nastavlja tamo gdje nema čekanja. Brzo prebacivanje konteksta zahtijeva odvojene skupove registara za različite niti iz istog procersa.
Četvrta metoda ubrzavanja - smanjivanja zastoja, je **neblokirajući upis u memoriju**. Ušteda u vremenu se postiže time da pricesor nakon pokretanja upisa u memoriju (instrukcije *store*) ne čeka na njen završetak već "ide dalje", dok posao završetka memorijskog ciklusa prepušta dodatnom hardveru.
Paralelni računari zahtijevaju "paralelni softver" - programe prilagođene izvršavanju na više procesora istovremeno. Slika 12.10. prikazuje neke od osnovnih metoda takve obrade podataka.



Slika 12.10. Osnovne metode paralelnog procesiranja

Protočna struktura (a) sa tri procesora dobija podatke u P1, čiji izlaz su podaci za P2 itd. Ako je niz podataka dovoljno dug (npr. obrada slike, video), svi procesori mogu biti istovremeno zauzeti.
Drugi metod je **podjela posla po fazama** (npr. iteracije petlje), kada svi procesori rade na jednom poslu, pa kada pojedini procesori završe, čekaju dok svi ostali završe, da bi se sinhronizovali i počeli izvršenje sljedeće faze (b).
Metodom **"podijeli pa vladaj"** (c) jedan proces pokreće/delegira nove procese koji se odrađuju sukcesivno i/ili istovremeno (analogno poslovima u građevinarstvu, kada glavni izvođač radova angažuje pod-izvođače po specijalnostima - zidre, električare, molere itd. koji mogu angažovati svoje pod-izvođače).
Red zadataka (farma) kao metoda počiva na centralnom redu čekanja zadataka iz kojeg procesori (radnici) uzimaju i odrađuju poslove (d). Ako zadatak generiše novi zadatak, taj novi se dodaje na kraj reda. Kada procesor završi jedan zadatak, uzima sljedeći iz reda čekanja. Ova metoda može uključivati i centralni procesor koji administrira dodjelu zadataka.

Kada se program podijeli na dijelove (procese) koji se izvršavaju paralelno, oni mogu međusobno komunicirati na dva načina - **dijeljenjem varijabli** (u zajedničkoj memoriji) i **razmjenom poruka** (preko veznih mreža). Obje metode se mogu, uz određene modifikacije, koristiti i u multiprocesorskim i u multiračunarskim sistemima. Obzirom na broj primalaca, poruke mogu biti upućene jednom odredištu (eng. point-to-point), svim odredištima (eng. broadcast) ili na samo neka odredišta (eng. multicast). Pored komuniciranja, paralelni procesi se trebaju međusobno **sinhronizovati**. Tipičan primjer je situacija kada jedan proces upisuje dijeljene podatke, dok ih drugi čita. Mehanizmom međusobnog isključenja potrebno je spriječiti čitanje sve dok se svi podaci ne upišu. Najčešće korištene metode za to su semafori i zaključavanje (privremeno) zajedničkih resursa.

Literatura

1. David A. Patterson, John L. Hennessy, "Computer Organization and Design – The Hardware / Software Interface", Fourth Edition, Elsevier, Inc., 2009.

2. John L. Hennessy, David A. Patterson, "Computer Architecture A Quantitative Approach", Fourth Edition, Elsevier, Inc., 2007.

3. Andrew S. Tanenbaum, "Structured Computer Organization", Fifth Edition, Prentice Hall, New Jersy, 2006.

4. Hesham El-Rewini, Mostafa Abd-El-Barr, "ADVANCED COMPUTER ARCHITECTURE AND PARALLEL PROCESSING", A John Wiley & Sons, Inc. Publication, 2005.

5. Sivarama P. Dandamudi, "Guide to RISC Processors for Programmers and Engineers", Springer Science+Business Media, Inc., 2005.

6. W. P. Petersen, P. Arbenz, " Introduction to Parallel Computing", Oxford University Press, 2004.

7. Linda Null, Julia Lobur, "The Essentials of Computer Organization and Architecture", Jones and Bartlett Publishers, 2003.

8. Behrooz Parhami, "Introduction to Parallel Processing Algorithms and Architectures", Kluwer Academic Publishers, 2002.

9. Thomas L. Floyd, “Digital Findamentals”, Prentice-Hall, 1997.

10. Vincent P. Heuring, Harry F. Jordan, “Computer Systems Design and Archtecture”, Addison Wesley Longman, Inc., 1997.

11. Les Freed, “The History of Computers”, Ziff-Davis Press, Emeryville, CA, 1995.

12. John F. Wakerly, “Digital Design Principles and Practices”, Prentice Hall, 1994.

13. Alan W. Show, “Logic Circuit Design”, Sounders College Publishing, 1993.

14. Eugene D. Fabricius, “Modern Digital Design and Switching Theory”, CRC Press Inc., 1992.

Sadržaj