PARALELNI RAČUNARSKI SISTEMI

Ova skripta je nastala sa idejom da se studentu maksimalno olakša spremanje ispita iz predmeta:

PARALELNI RAČUNARSKI SISTEMI
na elektronskom fakultetu u Nišu

Upravo zato, ovo nije kompletan udžbenik I treba ga shvatiti samo kao pomagalo za postizanje cilja tj. polaganje ispita.

Sve greške (štamparske I one semantičke-smisaone) prijavite slanjem mail-a na sledeću adresu:

Ime: Marko Miličić
Adresa: markomil@yahoo.com

Marko Miličić,
Aleksinac, 2003

UVOD

Paralelni računar je računar sa više procesora približno iste brzine izračunavanja. Radi tako što se neki problem rastavi na podprobleme i svaki procesor rešava po jedan problem. Na kraju se sva parcijalna rešenja skupe i formira se konačan rezultat. Jasno je da je ovako rešen problem potrošio manje vremena nego da je ceo problem rešavao jedan računar zbog toga što su se mnoga izračunavanja izvršavala **paralelno** (konkurentno) jedo s drugim. Dakle, paralelni računar troši manje vremena za obavljanje istog posla od sekvencijalnog računara zato što se koristi hardver koji u isto vreme (paralelno) obrađuje podatke, pa tako za kraće vreme dobijamo završen neki posao.

Podela paralelnih računara:

U odnosu na način upravljanja tj. u odnosu na to kako se upravlja radom paralelnog računara, oni se dele na:

1. *Von Newman*–ove računare (računari upravljani programskim tokom)
2. *Data Flow* (računari upravljani tokom podataka)
3. *Reduction* (Demand driven) (računari upravljani zahtevom)

Von Neuman–ovi računari:

Kod ovih računara redosled izvršavanja instrukcija je strogo definisan programskim tokom, odnosno, sadržajem programskog brojača (PC–a). Programski brojač definiše koja je sledeća instrukcija tj. određuje adresu na kojoj se nalazi sledeća instrukcija. Dakle, ovo je takozvana ***control driven*** arhitektura. Paralelni sistemi sastavljeni od ovakvih računara se dobijaju sprezanjem više jednoprocesorskih računara ovog tipa. Medjutim javlja se problem! Taj problem je u tome što kod njih postoji deljiva memorija[*] za podatke i programe, pa zbog toga može doći do konfliktnih slučajeva o kojima se mora voditi računa, što naravno povećava složenost sistema (hardvera), poskupljuje hardver, a samim tim i vreme potrebno da se neki problem reši. Ovom problematikom ćemo se uglavnom mi baviti. Tj. pokušavaćemo da na različiti načine anuliramo uticaje tih konflikata.

Data flow (Data driven) računari:

Kod ovih računara redosled izvršavanja instrukcija nije definisan programskim brojačem[*]. Izvršava se ona instrukcija kojoj su operandi dostupni. Zato se ova arhitektura zove ***data driven***. Ne postoji problem zajedničke memorije jer je nema, tj. svi podaci koji su potrebni nekoj instrukciji čuvaju se u samoj instrukciji. Kada jedna instrukcija formira rezultat, ona ga pošalje ostalim instrukcijama kojima je taj podatak potreban. Kada jedna instrukcija uzme podatak, on više nije raspoloživ za ostale instrukcije. Ovaj vid paralelizma se naziva *sitnozrnasti* paralelizam. Ne postoji odgovarajući mehanizam za povezivanje ovakvih računara. Dakle, ovi računari su čisto teorijski i pominju se samo u teorijskim analizama i razmatranjima o paralelizaciji računara.

Reduction (Demand driven) računari:

Kod ovih računara instrukcija se izvršava tek kada nekoj drugoj instrukciji zatreba njen rezultat. Ova izračunavanja se mogu posmatrati kao niz ugnježdenih izračunavanja. Sa izračunavanjima se kreće od najdublje ugnježdenog dela. Ova vrsta računara i uopšte princip funkcionisanja nije mnogo pogodan za numerička izračunavanja. Kao i u predhodnom slučaju, ni za ovaj tip računara ne postoji praktična realizacija. U praksi se koriste sam Von Neumano–ovi računari.

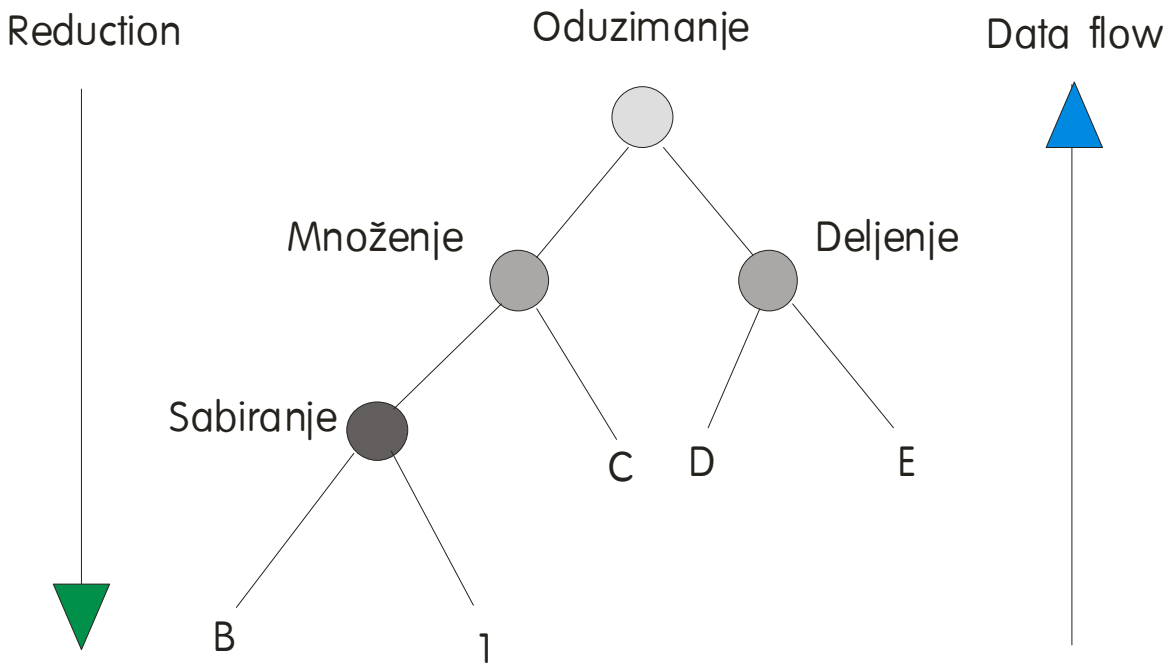
---

[*] I podaci i sam kod programa se nalaze u istoj memoriji, tako da kada se procesor obraća memoriji za neki podatak, ne može se u isto vreme obratiti memoriji za novu naredbu

[*] U ovom slučaju PC ne postoji

PRIMER:

Hajde da posmaramo sledeći izraz: $a = (b+1)\cdot c - \frac{d}{e}$ i njegovo izračunaravanje pomoću dataflow i reduction računara.



Data flow: Na osnovu raspoloživih podataka u prvom trenutku možemo izračunati b+1 i d/e. U sledećem trenutku (b+1)*c i u poslednjem ceo izraz a. Dakle, krećemo se od onoga što imamo da bi došli do onoga što nam treba. U Demand driven arhitekturi kretaćemo se od onoga šta nam treba i u odnosu na to izračunavati sve potrebne međukorake.

Demand driven: Kao što sam malopre rekao, nama treba a. Da bi smo to dobili tražimo razliku odgovarajućih čvorova sa grafa, a da bi se ona dobila treba nam množenje i deljenje odgovarajućih čvorova. A da bi se obavilo množenje, mora se pre toga izvršiti sabiranje elemenata b i 1. Očito da od ovog sabiranja počinje celo izračunavanje. Dakle, tok izvršavalja izračunavanja ovog izraza (a) tećiće sledećim tokom.

Reduction a=(((b+1)*c)–(d/e))

Von neuman–ovi računari i klasifikacija

Paralelni računari su mnogo puta klasifikovani. Od tih silnih klasifikacija, ustalila se danas najpopularnija klasifikacija, a to je **Flynn–ova klasifikacija**. Flynn je ovu klasifikaciju izvršio u odnosu na broj nezavosnih tokova (nizova) instrukcija i po njemu (Flynn–u) imamo 4 grupe paralelnih računara. Naravno, kao i u svakoj klasifikaciji ima i hibridnih varijanti koje su mešavine nekih osnovnih grupa.

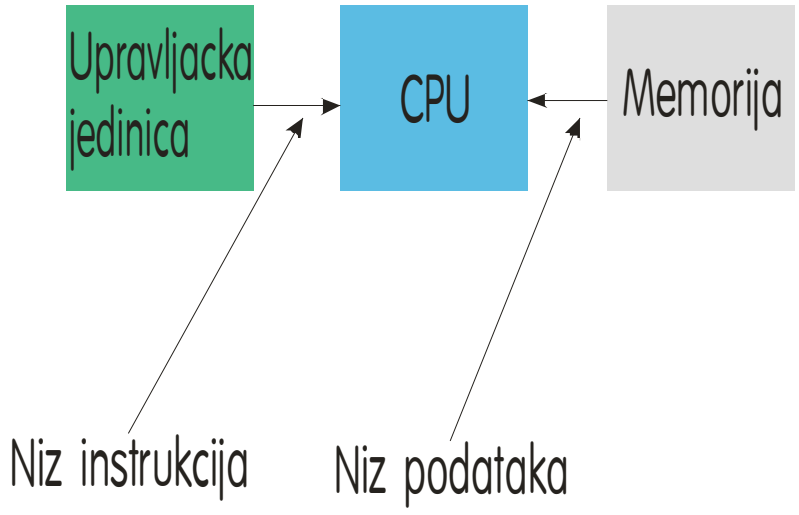
1. SISD **S**ingle **I**nstruction **S**ingle **D**ata stream
2. MISD **M**ultiple **I**nstruction **S**ingle **D**ata stream
3. SIMD **S**ingle **I**nstruction **M**ultiple **D**ata stream
4. MIMD **M**ultiple **I**nstruction **M**ultiple **D**ata stream

SISD računar (Single Instruction Single Data stream)

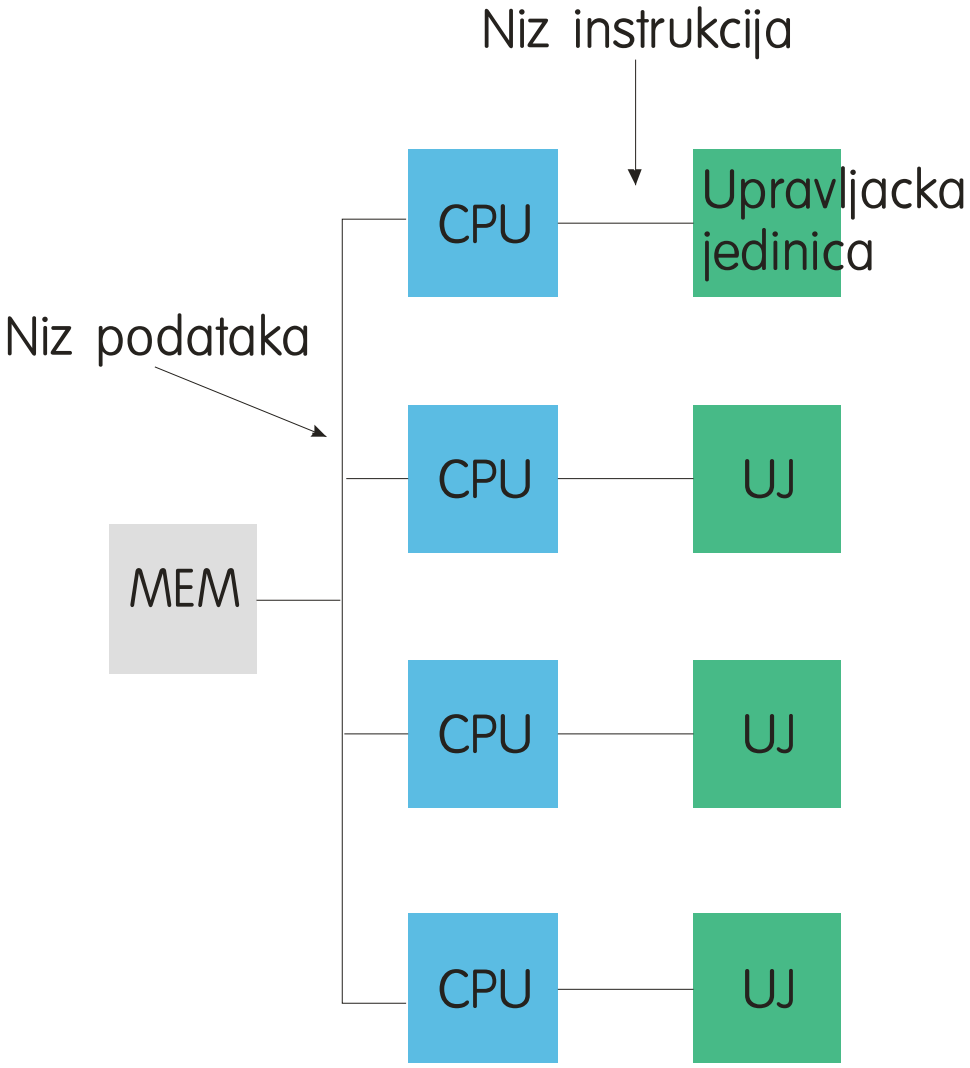
Ovo je klasičan jednoprocesorski računar sa jedinstvenim nizom podataka i jedinstvenim nizom instrukcija. Dakle, radi se o klasičnom **Von Neumanovom** računaru. Jasno je da ovde nema paralelizma, jer postoji samo jedna procesna jedinica. Naravno, za bilo kakav paralelizam potrebno je više od jedne procesorske jedinice (procesnog elementa).

Upravljačka jedinica izdaje niz instrukcija na osnovu kojih procesor radi sa nizom podataka koje dobija iz memorije.

Dakle, da zaključimo još jedared, ne postoji nikakav paralelizam jer postoji samo jedna procesna jedinica. Ovo je klasičan računar.



MISD računari



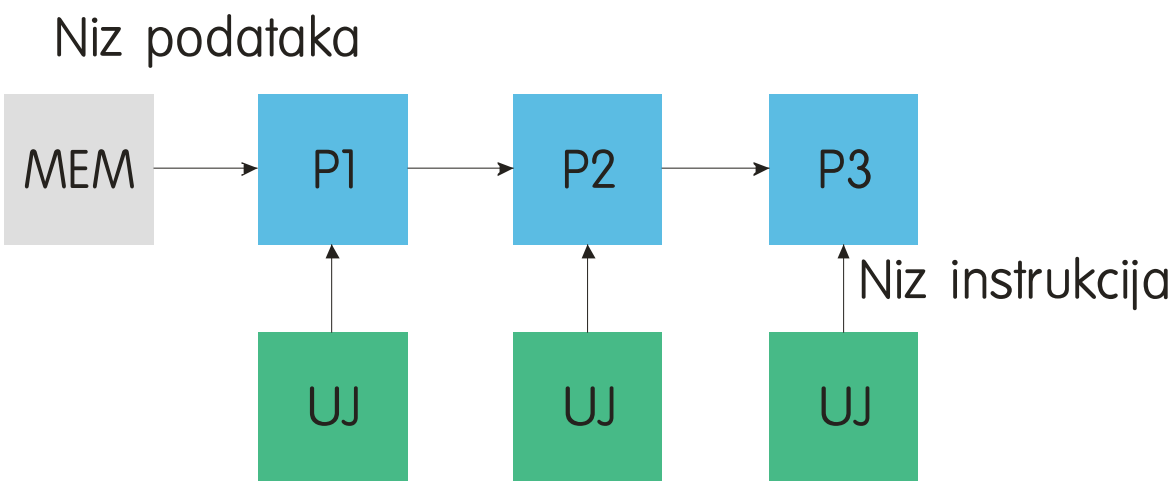
Ovo su sistemi kod kojih postoji višestruki tok instrukcija i jedinstveni tok podataka. Dakle, imamo nekoliko upravljačkih jedinica i samim tim nekoliko tokova instrukcija. Sistem ima N procesora i svaki od njih ima svoju upravljačku jedinicu, a svi koriste zajedničku memoriju. Dakle, svi procesori iz iste memorije dobijaju isti niz podataka (u datom trenutku svi dobijaju isti podatak koji treba da obrade) a svaki procesor iz svoje upravljačke jedinice dobija odgovarajući niz instrukcija, tako da se u istom trenutku nad istim podatkom u različitim procesorima vrše različite obrade.

MISD računar može se definisati i na drugi način. Na primer, kao protočni sistem na slici ispod teksta. Ovde se jedan niz podataka prenosi kroz protočni sistem i obrađuje. Svaki procesor u dobija niz izlaznih podataka prethodnog procesora, a prima niz instrukcija iz svoje upravljačke jedinice.

Ovako osmišljen režim rada nije povoljan za široku klasu aplikacija ali u nekim slučajevima je vrlo pogodan. Na primer, za prepoznavanje oblika (uzoraka) gde se testira posebno da li je neki oblik krug, trougao ili kvadrat. Tj. posebni procesorski elementi (u našem slučaju P1 P2 i P3) testiraju svako svoj oblik, tako da se značajno dobija na brzini zar ne?

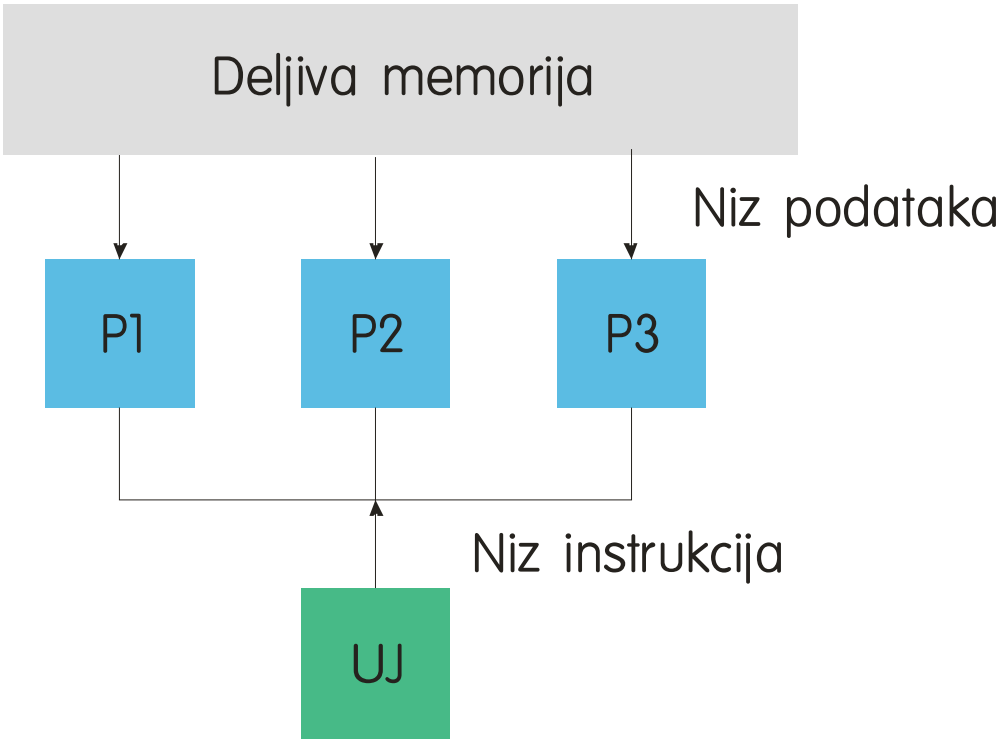
Podatak o slici tj. obliku koji se prepoznaje uzima se iz memorije i prosledjuje svim procesnim elementima. Ukoliko jedan od njih prepozna oblik, zadatak je završen.

Na sledećoj sistem prikazan je MISD računar kao protočni sistem.



Bez obzira na sve, MIMD i SIMD računari su mnogo korisniji za širu klasu aplikacija, te se zato u praksi više sreću.

SIMD računarai



Ovi sistemi imaju jedinstven niz instrukcija i višestruke nizove podataka. Postoji jedinstvena upravljačka jedinica koja upravlja radom svih N procesora. U jednom trenutku svi procesni elementi izvršavaju istu instrukciju (kao da svaki ima identičan program) ali nad različitim podacima. Ovi procesori mogu imati lokalnu memoriju u kojoj su smešteni podaci nad kojima procesor vrši obradu. Ovi podaci mogu biti različiti tj. najčešće su različiti jer bi bilo apsurdno računati isto izračunavanje da dva procesora. Tako da u suštini svaki procesni element izvršava istu instrukciju ali nad svojim podacima. Međutim, teško je izvršiti tako dobru podelu podataka (uvek je potrebna neka dodatna razmena podataka tj. sinhronizacija

i komunikacija među procesorima) i zbog toga treba da postoji veza između samih procesora. Ova veza se često realizuje kao **sprežna mreža** ili **deljiva memorija**.

SIMD računari rade u sinhronom režimu. Svi procesori rade sinhrono tj. svi u isto vreme obavljaju istu operaciju. Postoji mogućnost da u nekom trenutku neki procesori ne rade tj. da budu maskirani, ali ne i da postoji mogućnost da izvršavaju neku drugu instrukciju. Ovo je zgodno za realizaciju petlji.

Na primer, množenje matrica (svaki procesor izvršava operacije za različitu vrednost indeksa petlje). Ovi računari nisu pogodni za rešavanje problema koje nije moguće podeliti na niz identičnih problema (za njih se koriste MIMD računari).

Konkretan Primer: Množenje matrica

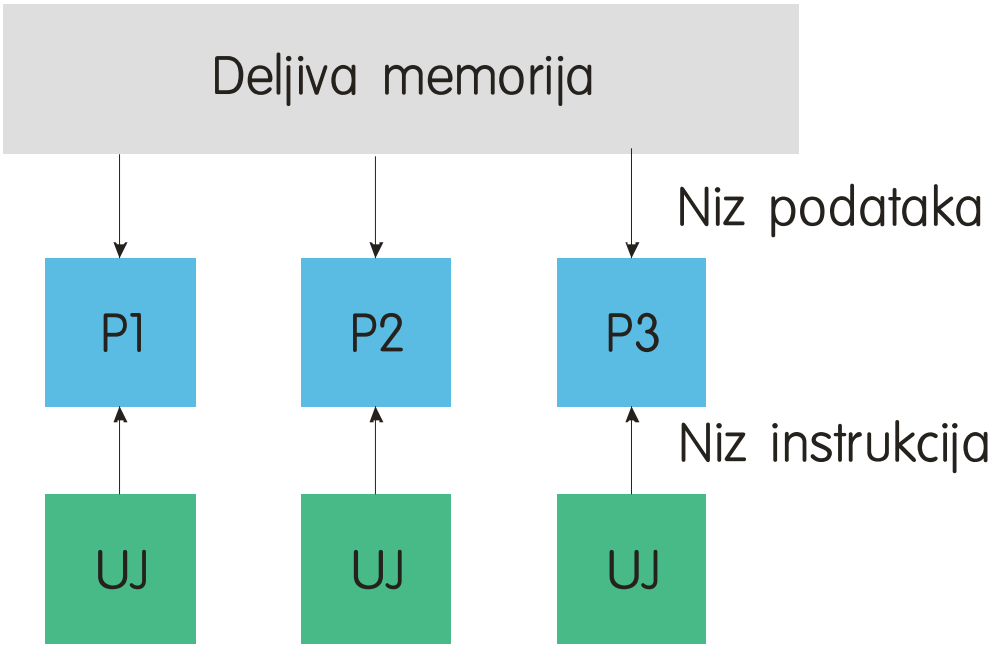
C=AxB

Znamo da je,

C(I,J) = C(I,J) + A(I,K) * B(K,J) ;gde su matrice A i B dimenzija mxn.

Za n^2 procesora u sistemu svaki procesor bi radio isto. Ako imamo manje, jedan procesor bi bio za jednu vrstu ili kolonu. Ovaj režim takođe nije primenljiv za širu klasu problema tj. nije zgodan za probleme koje nije moguće podeliti na više identičnih problema.

MIMD računara



Ovo su najopštiji i najmoćniji paralelni računari. Imaju višestruki niz podataka i višestruki niz instrukcija. U ovom slučaju, svaki procesor je faktički računar za sebe tj. ima svoju upravljačku jedinicu i svoju memoriju tako da on izvršava instrukciju koju mu prosledi njegova upravljačka jedinica nad podacima iz njegove lokalne memorije (apsolutno je samostalan). Vidimo da različititi procesori u istom vremenskom trenutku obavljaju različite operacije nad različitim podacima. Dakle, oni rade **asinhrono**. Neophodno je obezbediti mehanizam za **sinhronizaciju procesora**, kada je to neophodno. Pod sinhronizacijom se podrazumeva korektni redosled izvršenja instrukcija ili uzajamno isključivo pravo pristupa deljivoj memoriji ili uopšteno govoreći deljivim resursima.

Na osnovu toga kako se obavlja razmena podataka između procesora oni se dele na:

1. Čvrsto spregnute sisteme (multiprocesori)
2. Slabo spregnute sisteme (multiračunari)

Multiprocesori: Razmena podataka između procesora se ostvaruje korišćenjem zajedničkih ili deljivih promenljivih koje su zapamćene u zajedničkoj (deljivoj) memoriji. Ovoj memoriji pristupaju svi procesori tj. svi procesori imaju isti **adresni prostor**. Čvrsta spega odnosi se na postojanje istog adresnog prostora.

Multiračunari: Razmena podataka između procesora se obavlja eksplicitnim **slanjem poruka** kroz **spreženu mrežu**. U ovom slučaju ne postoji zajednička memorija, već svaki procesor ima svoju memoriju kojoj samo on može da pristupa. Pošto nema zajedničkog adresnog prostora za ovaj sistem se kaže da je slabo spregnut.

Rezime (zaključak)

U zavisnosti od aplikacije mi biramo arhitekturu koju ćemo koristiti. U smislu rešavanja problema sve 4 arhitekture su ekvivalentne (svaka će rešiti problem) ali razlika je u tome što se neki problemi lakše odnosno brže rešavaju na na određenim arhitekturama.

Paralelni računarski sistemi (Elektronski Fakultet – NIŠ)

Marko Miličić

PERFORMANSE RAČUNARSKIH SISTEMA

Da bi se postigle idealne (maksimalne moguće) performanse računarskog sistema treba da postoji idealno slaganje između mogućnosti mašine i programa. Mogućnosti mašine se mogu poboljšati korišćenjem bolje tehniologije, bržeg hardvera inoviranjem arhitekture računara ili boljim upravljanjem resursima.

Ponašanje programa je teže predvideti zato što ono zavisi od:

1. Vrste aplikacije koja se izvršava
2. Odabranog algoritma za rešenje nekog problema
3. Strukture podataka koja se koristi za smeštanje podataka
4. Mogućnosti (efikasnosti) programskog jezika
5. Tehnologije kompilatora
6. Od umešnosti programera

Najjednostavnija mera programskih performansi je **proteklo vreme** tj. **vreme odziva**. To je vreme koje protekne od izdavanja zahteva za obradu do njenog izvršavanja tj. vreme kada rezultat postane raspoloživ za dalju obradu. Brži je onaj koji za isti obim posla ima kraće vreme odziva. Vreme odziva se sastoji od sledećih vremena:

1. Procesorsko vreme (vreme izračunavanja tj. vreme koje procesor koristi za obradu podataka)
2. Vreme pristupa memoriji
3. Vreme pristupa diskovima
4. Vreme za rad sa U/I uređajima
5. Dodatno vreme za sistemske procedure

Ovo vreme se teško meri u multiprogramskom režimu rada zato što se procesor se bavi nekom drugom aktivnošću dok čeka na neku operaciju koja dugo traje, a pri tom ne zahteva rad samog procesora, Na primer, U/I operacija je jako duga operacija, a za vreme njenog izvršavanja procesor ne radi ništa. Zato se ovo vreme, za koje procesor ne radi ništa koristi za obavljanje nekih instrukcija nekog drugog programa. Time se dobija značajno poboljšanje performansi.

Ako se posmatra procesor u multiprogramskom režimu rada onda je idealno ponašanje da se za određeno vreme obavi što veći broj poslova, zar ne? Što je u suprotnosti sa korisnikovim željama, naime, on želi, i jedino primećuje, vreme odziva računara. Ova mera povećanja performansi naziva se povećanje **propusnosti** (troughput–a).

Procesorsko vreme: Većina današnjih CPU–a se taktuje signalom fiksne učestanosti. Ovaj signal je naziva **signal clock–a** i reda je nekoliko nano–sekundi.

$$f_{clk} = \frac{1}{T_{clk}}$$

Tclk je taktna perioda

Fclk je taktna učestanost

Procesorsko vreme TCPU možemo odrediti kao:

TCPU=NCPU * Tclk

Oznake koje su se pojavile u ovoj formuli su:

TCPU je procesorsko vreme potrebno za izvršavanje nekog programa

NCPU je ukupan broj procesorskih taktnih impulsa

Bilo bi bolje da umesto NCPU imamo broj instrukcija. Ali one različito traju, pa se uzima neka srednja vrednost. Za CISC je taj raspon perioda CLK–a za različite instrukcije vrlo veliki (5–120clk) za razliku od RISC arhitekture kod koje sve naredbe približno isto traju.

Potrebno je poznavati neki srednji broj taktova po instrukciji za datu arhitekturu (CPI). On se može odrediti tako što se prati izvršenje većeg broja programa u toku dužeg vremenskog perioda i da se pri tome odredi učestanost (Pi) pojavljivanja određenog tipa instrukcije (Ii) koje se izvršavaju za vreme (ti). Vreme ti je obično dato kao broj potrebnih ciklusa takta. Ako ima N različitih tipova instrukcija, onda je:

$$CPI = \sum_{i=1}^{N} t_i \cdot p_i$$

Ii je i–ta instrukcija

Pi je učestanost pojavljivanja instrukcije Ii u programu

ti je vreme izvršenja instrukcije

Ukoliko malo razmislimo uvidećemo da važi sledeća relacija.

$$T_{CPU} = CPI \cdot N_{\sum i} \cdot T_{clk}$$

$N_{\sum i}$ je ovde ukupan broj mašinskih instrukcija u datom programu.

TCPU treba da se smanji da bi sistem bio efikasniji. Ovo smanjenje je uslovljeno:

1. Smanjenjem Tclk, a to pak zavisi od tehnologije i organizacije hardvera.
2. Smanjenjem CPI, a to zavisi od seta instrukcija posmatrane arhitekture i organizacije hardvera (kod RISC procesora sa protočnom organizacijom CPI je 1–1.5 a kod CISC procesora 5–6 i više.
3. $N_{\sum i}$ zavisi od skupa instrukcija posmatrane arhitekture ali i od tehnologije kompilatora (od efikasnosti kompilatora – bolji kompilator generiše optimalniji tj. kraći paralelni kod zar ne ?)

Pored vremena odziva, koje je jedina prava mera performansi računarskog sistema, koriste se i druge mere kao što je MIPS (**M**ilion **I**nstructions **P**er **S**econd):

$$MIPS = \frac{N_{\sum I}}{T_{CPU} \cdot 10^6} = \frac{f_{clk}}{CPI \cdot 10^6}$$

Ova mera je uglavnom namenjena za procenu performansi procesora, a ne celog sistema. MIPS je kao mera za ocenu performansi pogodan jer je lako razumljiv za široki krug korisnika, ali ima niz nedostataka kao što su:

1. Broj MIPS–ova zavisi od seta instrukcija, posmatrane arhitekture .... Što čini veoma teškim poređenje arhitektura sa različitim setom instrukcija.
2. Zavisi i od programa koji se izvršava čak iako je reč o istoj arhitekturi.
3. Može da da lošiji rezultat za mašinu koja realno radi brže.

Kao ilustraciju neobjektivnosti MIPS–a kao performansne mere uzećemo sledeći primer.

Primer: Mašina sa koprocesorom za FP operacije i mašina koja to radi softverski.

1. Mašini sa koprocesorom je potrebno: 1 instrukcija ili 3 clk–a
2. Mašini bez koprocesora je potebno: 10 instrukcija ili 15 clk–a

$MIPS_1 = \frac{1}{3 \cdot 10^6} \rightarrow$ ova mašina je realno brža

$MIPS_1 = \frac{10}{15 \cdot 10^6} \rightarrow$ ova pak ima veći broj MIPS–ova !!!!!!

Ovaj problem nastaje zbog toga što smo upoređivali nešto što nije uporedivo. Da bi se prevazišao ovaj problem uveden je **relativni MIPS** kao mera performansi. Kod relativnog MIPS–a nova arhitektura se upoređuje sa nekom referentnom mašinom.

$$MIPS_{rel} = \frac{T_{ref}}{T_{oc}} + MIPS_{ref}$$

Tref je vreme izvršenja na referentnoj mašini

Toc je vreme izvršenja na mašini čije se performanse trenutno ocenjuju

MIPSref je broj MIPS–ova referentne mašine.

Kao referentna (1 MIPS mašina) koristi se VAX 11/780.

Ipak, i ovde nije dorečeno šta sme a šta ne sme da se menja. Koliko sme da bude stepen optimizacije kompilatora, koje opcije treba podesiti za vreme dok se izvršava program.

Pored MIPS–a i relativnog MIPS–a kao mera performanse koristi se i **MFLOPS** (**M**ilion **FP O**perations **P**er **S**econd).

$$MFLOPS = \frac{N_{FP}}{T_{CPU} \cdot 10^6}$$

Kao mera performansi ona se koristi isključivo za ocenu procesora kada on izvršava FP operacije (uglavnom kod naučno–tehničkih aplikacija). Van tog konteksta, MFLOPS se ne sme koristiti kao mera performansi niti procesora niti sistema u celini. Na primer, kod tekst editora i kompilatora, koji uopšte ne koriste operacije sa pokretnom zapetom nema smisla računati niti meriti ovu performansnu meru, zar ne? Broj MFLOPS–ova će biti nula ma koliko računar bio brz (zato što se uopšte i ne izvršavaju FP

operacije). Dakle, uglavnom se MFLOPS koristi kao mera performansi kod naučno–tehničkih aplikacija koje obilno koriste FP operacije.

Ove mere opisuju performanse procesora, a ne celog sistema. Za tu svrhu (ocenu performanski celog sistema) koriste se tzv. **BENCHMARK** programi. To su reprezentativni programi koji se koriste za generisanje realne slike o ponašanju sistema.

Prva grupa: Sistem se najbolje procenjuje korišćenjem realnog softvera tj. **realnih benchmark** programa i upoređivanjem rada na više mašina, ali malo korisnika je u stanju da uradi to. Tj. da oceni da li i koliko mu je neki računar brži od postojećeg ili od nekog referentnog.

Druga grupa BENCHMARK programa su **JEZGRA** (kerneli). To su delovi realnih programa, i služe kao standard za procenu performansi. Invertovanje matrica, nelinearne jednačine su tipični primeri za procenu sistema u naučno–tehničkim aplikacijama i primer je LINPAK (softvare za rešavanje sistema linearnih jednačina). BENCHMARK programi su uglavnom pisani na višimprogramskim jezicima.

Treća grupa BENCHMARK programa su **TOY BENCHMARK** programi. Oni su vrlo popularni i kratki (stotinak redova koda) NPR:

1. Seave of Erathosten (nalazi sve proste brojeve od 1 do zadatog broja n)
2. Quick sort (za brzo sortiranje nizova)
3. Puzzle

Četvrta grupa, takozvani **SINTETIČKI BENCHMARK** programi su najmanje pouzdani. Oni simuliraju učestanost pojavljivanja nekih instrukcija u programu za određenu oblast rada, pa simuliraju korišćenje različitih struktura podataka itd ... ovoj grupi pripadaju: WhiteStone, DryStone i Linpac.

Whitestone: koristi se za sistem koji radi sa podacima u pokretnom zarezu (FP). Prvobitno je bio pisan u ALGOLU, pa na PASCALU. Rezultat je bio prikazivan u Kilo Wips–ovima (KWIPS) tj. 1000 Whitestone operacija u sekundi.

Drystone: Rezultat je bio izražavan u KDIPS (1000 Drystone instrukcija u sekundi) a on se koristio za procenu performansi pri radu sa celobrojnim podacima

Linpak: Rešava se 100 jednačina sa 100 nepoznatih u jednostrukoj ili dvostrukoj tačnosti. Kao jedinica se koristio MFLOPS

Spec92 i Spec95 je mix svih testova. Oni se sastoje od otprilike 17 programa koji testiraju različite podsisteme nekog računara. Neke firme su namerno optimizovale svoje mašine za BENCHMARK programe. Tipičan primer je firma AMD (Advanced Micro Devices) koja je optimizovala svoje procsore za PC računare da pokazuju bolje performanse na Benchmark programima da bi imali više uspeha na svetskom tržištu.

Performanse paralelnih računraskih sistema

Da bi se dobila procena o ostvarenom poboljšanju kod paralelnih sistema usled uvođenja neke promene uvodi se pojam **ubrzanja S**. Ubrzanje je odnos vremena izvršenja programa na standardnoj mašini i vremena izvršenja na mašini sa nekim poboljšanjima (izmenanama).

$$S = \frac{T(1)}{T(n)}$$

$$S = \frac{T_{s\tan dardno}}{T_{poboljsano}}$$

Za paralelne sisteme ubrzanje se definišekao količnik vremena izvršenja programa na jednoprocesorskom sistemu i vremena izvršenja istog programa na paralelnom sistemu (n–to procesorskom sistemu), T(n).

Za vreme T(1) trebalo bi da se izabere vreme izvršenja najboljeg sekvencijalnog algoritma. Takav se teško paralelizuje dok oni slabiji mogu, pa je zbog toga često algoritam na n–procesorskom sistemu različit od algoritma na jednoprocesorskom sistemu.

Amdhalov zakon.

Amdal je ova istraživanja i samu formulaciju zakona dao 60–tih godina, kada je paralelna obrada bila još uvek čista teorija, tj. još uvek nije bio napravljen ni jedan paralelni računar. Ovaj zakon se odnosi na ocenu gornje granice ubrzanja koje se može postići uvođenjem paralelne obrade.

Amdal je zaključio da se vreme izvršenja programa na jednoprocesorskom sistemu T(1) može podeliti na dva dela. Tj. na vreme izvršenja dela programa koji se ne može paralelizovati i vrme izvršenja programa koji se može paralelizovati.

T(1)=Ts+Tp

Ts je vreme koje se ne može paralelizovati

Tp je vreme koje se može paralelizovati.

što se tiče paralelnog sistema, stvar je vrlo slična.

T(n)=Ts + Tp/Tn + T0(n)

Tp/Tn je vreme koje se može paralelizovati podeljeno na broj procesnih elemenata (ukoliko se takva podela može izvršiti.

T0(n) je dodatno vreme potrebno za sinhronizaciju i komunikaciju između procesa.

$$T_s = \alpha \cdot T(1)$$

Sve ovo važi samo pod uslovom da se deo programa koji se može paralelizovati može podeliti na n delova. U skladu sa današnjim znanjima o paralelnim računarskim sistemima, potpuno je opravdano postojanje $T_0(n)$ vremena, jer je nužno da se potroši malo vremena i na samu komunikaciju među procesnim elementima, ne bi li oni bili svesni celog procesa izračunavanja čiji deo izvršavaju.

Alfa je u ovom izrazu takozvana **AMDALOVA FRAKCIJA** koja može uzimati vrednosti između nule i jedinice. Ukoliko je AMDALOVA FRAKCIJA = 0 onda se radi o idealnom paralelizmu, a ukoliko je pak uzela vrednost 1 onda nema paralelizma.

$$S = \frac{T(1)}{T(n)} = \frac{\alpha(T_1) + (1-\alpha)T(1)}{\alpha \cdot T(1) + \frac{(1-\alpha)T(1)}{n}} = \frac{n}{1+(n-1)\alpha}$$

$$S(n) = \frac{n}{1+(n-1)\alpha}$$

Amdal je posmatrao i granični slučaj tj. kada broj procesora teži beskonačnosti i na osnovu tih istraživanja dobio je:

$$\lim_{n\to\infty} S(n) = \frac{1}{\alpha}$$

Posle ovih straživanja usledio je zaključak da bez obzira na to koliko ima procesora, ubrzanje koje se postiže paralelizmom je ograničeno delom programa koji se ne može paralelizovati. NPR: kada se 5% ne bi moglo paralelizovati, onda bi maksimalno moguće ubrzanje bilo 20x. Ovo je unelo sumnju o opravdanosti uvođenja parallnih računarskih sistema. Međutim, Amdal nije uzeo u obzir činjenicu da njegova frakcija nije konstantna veličina. Dakle, (alfa) AMDALOVA FRAKCIJA zavisi i od obima problema tj. nije konstantna već se menja u zavisnosti od obima problema.

$$\alpha = \alpha(m)$$

Kod većih problema alfa teži nuli. Ovo zapravo znači da je deo programa koji se ne može paralelizovati sve manji kada obim problema raste.

$$\lim_{n\to\infty} \alpha(m) = 0$$

$$\lim_{n\to\infty} S(n) = \lim_{m\to\infty} \frac{n}{1+(n-1)\alpha(m)}$$

Kada obim problema teži beskonačnosti (ogromni problemi) ubrzanje postaje linearno zavisno od broja procesora. Za algoritme koji imaju osobinu da im se ubrzanje ponaša linearno sa porastom obima problema kažemo da su **EFEKTIVNI PARALELNI ALGORITMI.**

Na Primer:

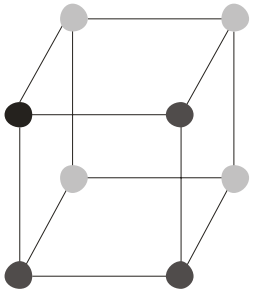
Jedan takav algoritam može se napraviti za n–procesoski paralelni sistem sa topologijom **hiperkuba** za rešavanje problema:

Y=A+X A mxm, X mx1

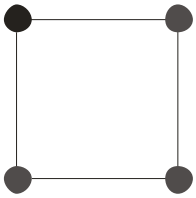
Dakle, za nalaženje vektora Y mx1 kao proizvoda matrice A i Vektora X odgovarajućih dimenzija.

Ove matrice treba pomnožiti na n–procesorskom sistemu. Tj. na n–procesorskoj hipper–kocki (hyper cube). Hiperkocka je sistem sa $n=2^r$ procesora (stepen dvojke). Veze između procesora su za r=3 $2^3 = 8$ procesora, pa su oni raspoređeni u temenima kocke. Svaki procesor ima svoju lokalnu memoriju. Dakle, radi se o multiračunarima. Suprotno ovome multiprocesori imaju slanje poruka kao vid komunikacije.

Svaki procesor je direkno povezan sa r najbližih suseda. **Maksimalno rastojanje** između bilo kog para procesora u r dimenzionalnom kubu je r koraka $r=\log_2 n$.



R=3



R=2

Neka se u memoriji jednog elementa nalazi zapamćen vektor X, a u memoriji svih procesora se nalazi $k=\left[\frac{m}{k}\right]+1$ po k vrsta matrice A.

Ukoliko ima ostatka neki procesni elementi će imati jednu vrstu više.

Efektivni paralelni algoritam za množenje matrice vektorom na datom sistemu bi se mogao predstaviti preko nekoliko koraka. Dakle, koraci bi bili:

1. Distribucija vektora X svim procesorima. Svaki ima po parče matrice i ceo vektor X.
2. Svaki procesor može izračunati po K elemenata vektora Y po formuli:

$$Y_i = A_i \cdot X = \sum_{j=1}^{n} a_{ij} \cdot x_j \quad \text{i=1,k} \qquad \text{j=1,n}$$

3. Procesori šalju podatke recimo u procesor P_1 i na kraju ćemo imati gotov rezultat u procesoru P_1.

Sada treba da posmatramo koliko vremena traje sve ovo, ako pretpostavimo da jedna operacija sabiranja ili množenja traje jednu vremensku jedinicu.

Dakle, postavlja se pitanje koliko vremena je potrebno za sve to?

1. Da bi se jedan element preneo iz P1 u neki procesor, najduži put koji treba da pređe je $\log_2 n$. Pošto treba distribuirati m elemenata, cela ova aktivnost traje

   $O(m \cdot \log_2 n)$ m je broj elemenata a $\log_2 n$ je potrebno vreme za 1 element

2. Da bi se jedan element izračunao treba obaviti m množenja i m–1 sabiranje, dakle, red veličine problema je O(m). Svaki procesor izračunava K elemenata, pa je cela ova aktivnost O(km)

$$O(m \cdot k) = O\left(\frac{m^2}{n}\right) + O(m)$$

3. Ovaj korak je obrnut prvom koraku, a zahteva isto vremena

$$O(m \cdot \log_2 n)$$

Nas inače, zanima koji od ovih elemenata utiču na Ambalovu frakciju

$$\alpha = \frac{T_s}{T(1)} \Rightarrow T_s = \alpha \cdot T(1)$$

Na jednoprocesorskom sistemu za množenje vektora i matrica treba izračunati m elemenata, a za izračunavanje svakog elementa potrebno je vreme reda veličine O(m) pa je $T(1)=O(m^2)$.

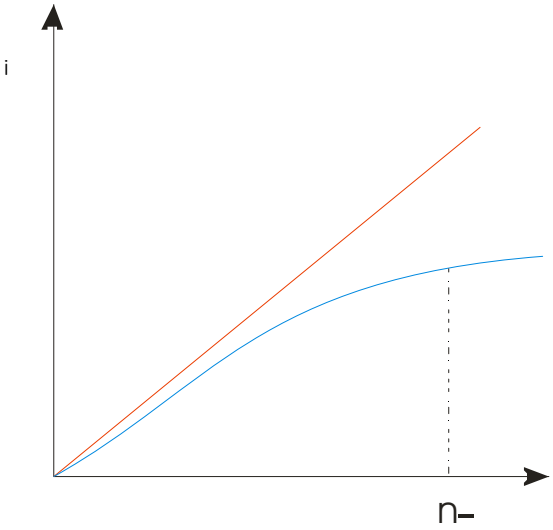
$$T(1) = O(m^2)$$

$$T(n) = O(m \cdot \log_2 n) + O(m)$$

$$\alpha = \frac{T_s}{T(1)} = \frac{O(m \cdot \log_2 n) + O(m)}{O(m^2)} = O\left(\frac{1+\log_2 n}{m}\right)$$

$$\lim_{m\to\infty} \alpha = \lim_{m\to\infty} O\left(\frac{1+\log_2 n}{m}\right) = 0$$

Dakle, ovo je zaista efektivni paralelni algoritam.



Međutim, ubrzanje nije baš linearno već od neke tačke ako je m konstantno ubrzanje prestaje linearno da raste počinje da opada.

T(n)=Ts + Tp/Tn + T0(n).

U ovom obrazcu sa T0(n) prestavljeno je dodatno vreme za sinhronizaciju procesa. Drugim rečima, sa porastom n T0(n) postaje dominantno vreme pa se javlja nepravilnost u linearnom rastu koju smo malopre opisali. Sa porastom broja procesora u sistemu T0(n) to vreme postaje dominantno tj. procesor više vremena troši na komunikaciju nego na obradu. Ako imamo više procesora u sistemu nego što je potrebno, onda je ekonomičnije da one koji su nam višak odbacimo ili zaposlimo da rade neki drugi posao.

Pored ubrzanja važna mera performansi je i **efikasnost sistema**. Efikasnost se definiše kao količnik postignutog ubrzanja i broja procesora u sistemu.

$$E(n) = \frac{S(n)}{n}$$

U ovom obrascu sa S(n) je obeleženo ubrzanje sistema o kome smo do malopre govorili, a sa n je obeležen broja provesora. Kao mera performansi, efikasnost nam govori kolika je srednja iskorišćenost procesora u sistemu. Može se smatrati da je efikasnost jednoprocesorskog sistema E(1)=1, ako se zanemare ulazno–izlazne aktivnosti. Sa porastom broja procesora efikasnost opada. Poželjno je da efikasnost ne bude manja od 50%.

Kao mera efikasnosti, efikasnost govori kolika je srednja iskorišćenost procesora u sistemu i kreće se između nule i jedinice. Ako se uzmu U/I aktivnosti može da se uzme da je E(n) za jenoprocesorski sistem jednako jedinici. Sa porastom broja procesora efikasnost opada i poželjno je da ona ne bude

manja od 50%. Realno, gde je n0 (maksimalno ubrzanje tu je i efikasnost manja od 50% naravno treba biti vešt i odabrati optimalno rešenje. Upravo zbog toga posmatramo proizvod ove dve mere:

F(n)=S(n)E(n)

I traži se maksimum tog proizvoda.

Na primer:

Treba naći sumu S=a1+a2+a3+.....+a16

Koliko bi vremena bilo potrebno za:

n=2 procesora ?

P1	P2
B1=a1+.....a8	B2=a9+.....+a16
S=b1+b2	

Dakle, T(2)=8

Tada bi procesor P1 računao b1=a1+.....a8 a procesor P2 bi za to vreme računao b2=a9+.....+a16. Krajnji rezultat je sada S=b1+b2. Dakle, T(2)=8

n=3 procesora ?

P1	P2	P3
B1=a1+.....a5	B2=a6+.....+a10	B3=a11+.....a15
C1=b1+b2	C2=b3+a16	
S=c1+c2		

Dakle, T(3)=6

Tada bi procesor P1 računao b1=a1+.....a5 a procesor P2 bi za to vreme računao b2=a6+.....+a10 a procesor P3 bi ražunao b3=a11+.....a15. Kada se završi ovaj prvi korak procesori P1 i P2 će sada računati zbirove C1=b1+b2 i C2=b3+a16. Krajnji rezultat je sada S=c1+c2. Dakle, T(3)=6

n=4 procesora?

P1	P2	P3	P4
B1=a1+...+a4	B2=a5+....+a8	B7=a9+....+a12	B5=a13+.....+a16
C1=b1+b2	C2=b3+b4		
S=c1+c2			

Dakle, T(4)=5

n=8 procesora?

P1	P2	P3	P4	P5	P6	P7	P8

b1=a1+a2	B2=a3+a4	B3=a5+a6	B4=a7+a8	B5=a9+a10	B6=a11+a12	B7=a13+a14	B8=a15+a16
C1=b1+b1	C2=b3+b4	C3=b5+b6	C4=b7+b8				
D1=c1+c2	D2=c3+c4						
S=d1+d2							

Dakle, T(4)=5

Sada možemo napraviti pregled u vidu tabele.

	T(n)	S(n)	E(n)	F(n)
2	8	1.875	0.94	1.76
3	6	2.5	0.83	2.0
4	5	3	0.75	2.25
8	4	3.75	0.47	1.7

Pojava da se iz koraka u korak broj aktivnih procesora smanjuje zove se **HAJDNOV EFEKAT**. Dobio je ime po velikom kompozititoru koji je tako pisao svoja dela u nameri da poštedi muzičare i odmara ih za narednu kompoziciju.

Upravo zbog toga je bolje projektovati elgoritam za paralelnu obradu nego paralelizovati serijski algoritam zar ne ?

ZADACI (PERFORMANSE SISTEMA)

Rekapitulacija Teorije

Za korisnika je računar brz ako mu izvrši što više zadataka za što kraće vreme. To je vreme odziva. Stoga se performanse računara ne mogu apsolutno odrediti jer zavise od softvera i hardvera.

Kod rešavanja većine problema projektovanje komponenti računarskog sistema (kompajleri itd ...) moguće je neke parametre poboljšati samo ako se neki drugi pogoršaju. Opšti princip je da treba optimizirati parametre sa većim uticajem na račun drugih čiji je uticaj na ukupne perforanse manji. Ove zaključke opisuje takozvani Amdalov zakon.

Amdalov zakon: Poboljšanje performansi koje se postiže uvođenjem nekog bržeg načina rada ograničeno je količinom vremena u kome se taj način rada koristi. Ovako definisan zakon ima opšti karakter i njime se definiše ubrzanje nekog sistema koje se može postići poboljšanjem neke njegove karakteristike.

$$\text{ubrzanje} = \frac{\text{performanse sistema sa poboljsanjem}}{\text{performanse sistema bez poboljsanja}} = \frac{\text{vreme sistema bez poboljsanja}}{\text{vreme sistema sa poboljsanjem}}$$

Performanse višeprocesorskog sistema

Amdalov zakon predstavlja ograničenje za ubrzanje koje se postiže kod višeprocesorskih sistema. Za razmatranje ovog ograničenja potrebne su definicije pojmova ubrzanje i efikasnost. Neka se neki program izvršava na p–procesorskom sistemu za vreme $T_p(n)$ gde je n red veličine složenosti problema. Za uvođenje pojmova ubrzanje i efikasnost potrebno je da se zna za koje vreme se određeni program rešava na jednoprocesorskom sistemu. To vreme se označava sa $T_1(n)$. ovo vreme je teško odrediti jer i za najstandardnije probleme ne postoje algoritmi za koje se sa sigurnošću može reći da daju minimalno $T_1(n)$. Alternative za izbor $T_1(n)$ su:

- $T_1(n)$ je vreme izvršenja najboljeg postojećeg serijskog algoritma.
- $T_1(n)$ je vreme koje se dobija kao rezultat BENCHMARK programa.
- $T_1(n)$ je vreme potrebno jednoprocesorskom sistemu da izvrši paralelni algoritam.

Ubrzanje:

Ubrzanje P–procesorskog sistema je:

$$S_p(n) = \frac{T_1(n)}{T_p(n)}$$

To je odnos vremena izvršenja algoritma za 1 procesor i p procesora. Idealno bi bilo kada bi bilo:

$$S_p(n) = P$$

Efikasnost:

Efikasnost sistema označava u kojoj se meri pojedinačni procesor korisno upotrebljava. Definiše se kao:

$$E_p(n) = \frac{S_p(n)}{p}$$

Dakle, efikasnost je odnos ubrzanja p procesorskog sistema i broja procesnih elemenata.

Ako uvedemo oznaku α za deo sekvencijalnog algoritma koji se ne može paralelizovati tj. koji se mora izvršavati serijski, onda še vreme izvršavanja n p–procesorskom sistemu da bude:

$$T_p = \alpha \cdot T_1 + \frac{(1-\alpha)\cdot T_1}{p}$$

Ubrzanje na p–procesorskom sistemu je tada:

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{\alpha \cdot T_1 + \frac{(1-\alpha)\cdot T_1}{p}} = \frac{p \cdot T_1}{\alpha \cdot p \cdot T_1 + (1-\alpha)\cdot T_1} = \frac{p}{\alpha \cdot p + 1 - \alpha}$$

Ovo je izraz za Amdalov zakon za višeprocesorski sistem.

$$\lim_{p\to\infty} S_p = \frac{1}{\alpha}$$

Dakle, nezavisno od broja procesora ubrzanje zavisi od faktora α tj. od onog dela programa koji se ne može paralelizovati (ubrzati).

$$\lim_{n\to\infty} \alpha(n) = 0$$

Ovo se može pročitati kao:

Za veliki obim posla ima smisla koristiti paralelni sistem, u protivnom je neekonomično!

$$\lim_{n\to\infty} S_p = \lim \frac{p}{1+(p-1)\alpha(n)} = p$$

Zadatak br. 1

Putnik putuje od Leskovca do Beograda pri čemu ima na raspolaganju:

- Pešačenje brzinom 4km/h
- Vožnju bicikle brzinom 10km/h
- Vožnju fićom brzinom 50km/h
- Vožnju golfom brzinom 120km/h
- Vožnju avionom brzinom 600km/h

Prvih 20km do Niša mora preći pešice, dok mu sva navedena prevozna sredstva mogu koristiti tek od Niša u narednih 200 km. Koliko mu je vreme potebno za prelaženje ovog puta korišćenjem navedenih sredstava i koliko je ubrzanje u odnosu na slučaj kada bi ceo put prešao pešice?

Resenje zadatka br. 1

Prevozno sredstvo	Časovi potrebni za prelazak drugog dela puta.	Ubrzanje na putu NI–BG	Časovi potrebni za prelazak celog puta.	Ubrzanje za ceo put
Pešice	200/4=50	1	50+20/4=50+5=55	1
Bicikl	200/10=20	50/20=2.5	20+5=25	55/25=2.2
Fića	200/50=4	50/4=12.5	4+5=9	55/9=6.11
Golf	200/120=1.67	50/1.67=30	1.67+5=6.67	55/6.67=8.25
Avion	200/600=0.33	50/0.33=150	0.33+5=5.33	55/5.33=10.32

Sasvim je logično da je broj časova potrebnih da se pređe drugi deo puta jednak količniku preostalog puta (drugog dela puta) i brzine prevoznog sredstva koje se koristi. Dakle, ukoliko bi i dalje išli peške, bilo bi nam potrebno 50 sati, ukoliko bi se kretali biciklom bilo bi nam potrebno 20 sati, i tako dalje. Ono što je zanimljivo za analizu ubrzanja je to, koliko je ubrzanje na drugom delu puta u odnosu na slučaj kada bi se i u drugom delu puta kretali pešice. O tome nam govori druga kolona tabele. Ukoliko bi se i u drugom delu puta kretali pešice, ne bi bilo nikakvog ubrzanja tj. ubrzanje bi bilo jednako jedinici. Ukoliko bi smo se pak kretali biciklom, ubrzanje bi bilo 50/20. Ovo je zato što je 50 broj sati koji nam je potreban za prelaženje drugog delog puta pešice, a 20 brojsati potrebnih za prelaženje drugog dela puta biciklom. Logično je da je ovaj odnos potrebnih vremena upravo ono što nazivamo *ubrzanje*.

Ukoliko bi trebali da izračunamo broj sati potrebnih za prelaženje celog puta, onda bi to bilo prikazano trećom kolonom u tablici. Dakle, na broj sati potrebnih za prelaženje drugog dela puta, dodaćemo broj sati potrebnih da se pređe prvi deo puta (prvi deo puta je 20 km i prelazi se pešice, dakle, to je 5 sati). Sledeći ovu logiku, dobićemo treću kolonu tabele.

A sada, treba izračunati ubrzanje koje će putnik postići za ceo put. Tj. ukupno gledajući, koliko će mu vremena manje trebati za ceo put, ako se on kreće sa svakim od navedenih prevoznih sredstava umesto pešice.

Ovo ubrzanje dobićemo tako što podelimo vreme koje bi trebalo da prođe da bi se prešao put u celosti pešice, sa vrmenom koje je potrebno za prelaženje tog puta koristeći neko od poboljšanja.

Iz tabele (četvrta kolona) vidi se da je ukupno ubrzanje, kada bi on nastavio pešice 1 tj. ubrzanja nema. Ubrzanje na celom puta, kada bi on koristio biciklu na drugom delu puta bilo bi 2.2 i tako dalje.....

Amdalov zakon daje način za određivanmje ubrzanja koje zavisi od 2 faktora.

1. Deo vremena na originalnom sistemu u kome se može koristiti poboljšanje je: $f_{re} = \frac{50}{55}$ za ovaj zadatak. Ovo vreme je uvek manje od jedinice. Ukoliko bi bilo veće od jedinice onda bi to značilo da se ceo sistem promenio, te više ne govorimo o tom sistemu već o nekom drugom (boljem).
2. Dobitak usled uvođenja poboljšanja, f_{pe}. U ovom slučaju/zadatku to je ubrzanje na putu od Niša do Beograda. Ova veličina je, za razliku od prethodne, uvek veća od jedinice.

Hajde sada da sa E_{tn} označimo neko novo vrme izvršavanja, a sa E_{t0} staro vreme izvršavanja, a sa f_{pu} označićemo ukupno ubrzanje. Tada bi smo imali da važi:

$$E_{tn} = E_{t0}\left((1 - f_{re}) + \frac{f_{re}}{f_{pe}}\right) \qquad f_{pn} = \frac{1}{(1 - f_{pe}) + \frac{f_{re}}{f_{pe}}}$$

Zadatak br. 2

Posmatramo mašinu LOAD/STORE tj. mašinu koja podatke sa memorijom razmenjuje samo preko LOAD i STORE naredbi, a sve ostale operacije rade sa registrima. Neka je skup naredbi ovakve mašine opisan tablom:

Operacija	Učestanos	Broj taktova potrebnih za izvršavanje
ALU	43%	1
LOAD	21%	2
STORE	12%	2
BRANCH	24%	2

Neka 25% ALU operacija koriste neposredno pre toga učitan operand tj. operand u memoriji pre korišćenja operacije. Uvodimo modifikaciju mašine. Dodali smo ALU instrukcije koje imaju jedan operand u memoriji. Ove instrukcije imaju 2 taktna perioda i povećavaju broj taktnih perioda naredbe grananja za 1, te one sada imaju 3 taktna perioda. Učestanost pojavljivanja ovih instrukcija ostaje nepromenjena.

Da li su ovim performanse poboljšane ili pogoršane ?

Resenje zadatka br. 2

$$CPI = \frac{\text{Broj klok ciklusa u programu}}{\text{Broj instrukcija u programu}}$$

Dakle, ovo je broj ciklusa kloka potrebnih za jednu instrukciju! I to njegova prosečna vrednost.

Skraćenice koje mogu biti vrlo korisne ukoliko se zna šta znače:

CP	Taktni period (clock period)
CPUT	Vreme za koje se koristi CPU (CPU Time)
IC	Broj instrukcija (Instruction Count)

Sistem bez poboljšanja:

$$CPI_0 = \frac{0.43 \cdot I_{C0} \cdot 1 + 0.21 \cdot I_{C0} \cdot 2 + 0.12 \cdot I_{C0} \cdot 2 + 0.24 \cdot I_{C0} \cdot 2}{I_{C0}} = 1.57$$

Dakle, CPI_0 je prosečno trajanje instrukcije (računato u ciklusima takta) u sistemu bez poboljšanja.

Ukupno vreme potrebno za izvršavanje svih instrukcija se dobija kada se pomnoži:

- IC (broj instrukcija u progranu)
- Prosečno trajanje jedne instrukcije
- Dužina klok signala

$$CUPT_0 = IC_0 \cdot CPI_0 \cdot CP_0 = 1.57 \cdot IC_0 \cdot CP_0$$

Jasno je da proizvod ove tri veličine izražava ukupno vreme potrebno za izvršenje celog programa.

Sistem sa poboljšanjima:

$$CPI_n = \frac{(0.75 \cdot 0.43) \cdot I_{C0} \cdot 1 + (0.21 - 0.25 \cdot 0.43) \cdot I_{C0} \cdot 2 + 0.25 \cdot 0.43 \cdot I_{C0} \cdot 2 + 0.12 \cdot I_{C0} \cdot 2 + 0.24 \cdot I_{C0} \cdot 3}{I_{Cn}} = 1.7025 \frac{I_{c0}}{I_{cn}}$$

Šta se ove sada dogodilo?

Ono što je sigurno, je da su se sada ALU operacije podelile na dva dela koja sadrže 25% i 75% od početne učestanosti ALU naredba. Međutim, pošto 25% naredba ALU neće koristiti LOAD neredbu, onda je i učEstanost LOAD naredbi umanjena za četvrtinu iznosa (25%) ALU naredbi. Ovo je sve prikazano u prva tri člana sume koja računa CPI_n.

Dakle, CPI_n je prosečno trajanje instrukcije (računato u ciklusima takta) u sistemu sa poboljšanjima.

Ukupno vreme izvršenja programa je, ovog puta:

$$CPUT_n = IC_n \cdot CPI_n \cdot CP_u = IC_n \cdot 1.7025 \cdot \frac{IC_0}{IC_n} \cdot CP_0 = 1.7025 \cdot IC_0 \cdot IP_0$$

Pošto je $CP_n = CP_0$ (učestanost je ostala nepromenjena) zaključujemo da su se uvođenjem modifikacije pogoršale performanse, jer je CPUT povećano.

Zadatak br. 3

Projektuje se optimizirajući kompajler za LOAD/STORE mašinu iz prethodnog zadatka i to pre modifikacije. Neka kompajler smanji broj ALU instrukcija za 50%, a broj ostalih instrukcija ostaje isti. Ako je taktni period 20 ns kolike su performanse u oba slučaja.

Rešenje zadatka br. 3

Performanse ćemo izraziti u MIPS–ima!

Sistem bez poboljšanja:

$$MIPS = \frac{\text{broj instrukcija}}{\text{vreme izvrsenja} \cdot 10^6} = \frac{IC}{CPUT \cdot 10^6} = \frac{IC}{CPI \cdot IC \cdot CP \cdot 10^6} = \frac{\frac{1}{CP}}{CPI \cdot 10^6}$$

CPI_0=1.57 (ovo smo izračunali u prošlom zadatku, tako da neću duplirati izračunavanja.)

$$MIPS_0 = \frac{\frac{1}{CP}}{CPI_0 \cdot 10^6} = 31.85$$

$$CUPT_0 = IC_0 \cdot CPI_0 \cdot CP_0 = 31.4 \cdot 10^{-9} \cdot IC_0$$

Sistem sa poboljšanjima:

Prvo treba izračunati koliko je sada instrukcija ukupno u programu s obzirom na to da je broj ALU instrukcija smanjen za 50%.

Dakle,

$$IC_n = IC_0(1 - 0.43 \cdot 0.5) = 0.785 \cdot IC_0$$

Sada, izračunavamo prosečno trajanje instrukcije u novom sistemu.

$$CPI_n = \left[(0.43 - 0.5 \cdot 0.43) \cdot 1 + 0.21 \cdot 2 + 0.12 \cdot 2 + 0.24 \cdot 2\right] \cdot \frac{IC_0}{IC_n} = 1.355 \cdot \frac{1}{0.785} = 1.73$$

dakle, ako kao meru performansi uzmeno MIPS–ove (ponovo) dobićemo:

$$MIPS_n = \frac{\frac{1}{CP}}{CPI_n \cdot 10^6} = 28.9$$

Dok je ukupno vreme potrebno da se program izvrši:

$$CUPT_n = IC_n \cdot CPI_n \cdot CP_n = 0.785 \cdot IC_0 \cdot 1.73 \cdot 20 \cdot 10^{-9} = 27.2 \cdot 10^{-9} IC_0$$

Šta se odavde može zaključiti ?

Zaključujemo da je novi sistem brži jer je ukupno vreme izvršenja kraće, ali javlja se jedna anomalija !

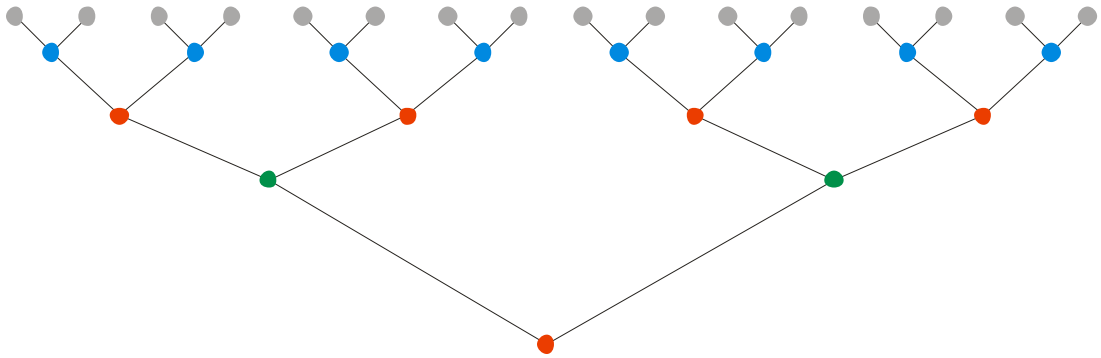
Naime, ukoliko bi smo uzeli MIPS–ove kao meru performansi onda bi nam ispalo da novi sistem ima manji broj MIPS–ova ?

Odavde zaključujemo MIPS nije objektivna mera performansi nekod sistema. To smo kazali i na predavanjima zar ne?

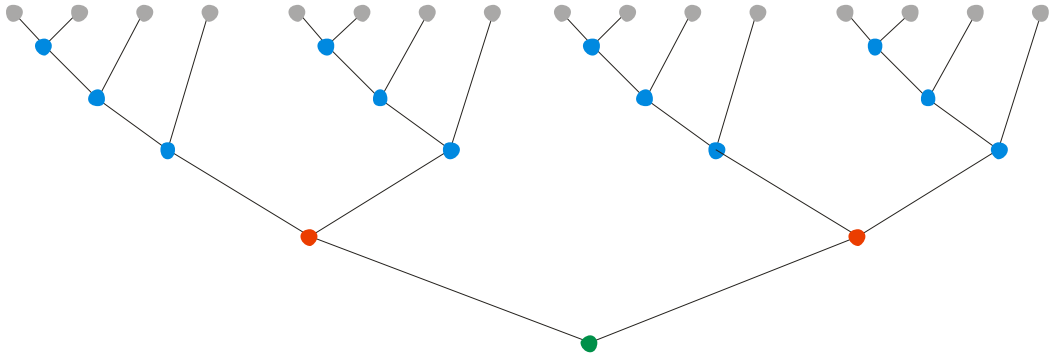
Zadatak br. 4

Na slikama 1 i 2 prikazana su 2 algoritma za sabiranje n skalara (n=16), gde je n neki stepen dvojke. Odrediti ubrzanje i efikasnost za date algoritme ako se oni izvršavaju na višeprocesorskom sistemu.

Slika 1



Slika 2



Rešenje zadatka br. 4

Prvo ćemo analizirati sliku1:

Vidimo da se ovde koristi $\left\lfloor \frac{n}{2} \right\rfloor$ procesora za sabiranje skalara. U sledećem koraku se sabiraju po dva rezultata iz predhodnog koraka. Ukupan broj koraka je: $\lceil \log_2 n \rceil$.

Za izvršenje najboljeg serijskog algoritma na jednoprocesorskom sistemu potrebno je n–1 operacijajer da bi sabrali 5 brojeva, treba nam najviše četri operacije sabiranja zar ne ?

Dakle:

$$T_1(n) = n - 1$$

$$T_p(n) = \lceil \log_2 n \rceil$$

Iz ovoga možemo da izvedemo ubrzanje sistema:

$$S_p^1 = \frac{T_1(n)}{T_p^1(n)} = \frac{n-1}{\lceil \log_2 n \rceil}$$

efikasnost sistema se takođe može izračunati kao:

$$E_p^1(n) = \frac{S_p^1(n)}{\left\lfloor \frac{n}{2} \right\rfloor} = \frac{n-1}{\left\lfloor \frac{n}{2} \right\rfloor \cdot \log_2 n}$$

Ako stavimo da n teži beskonačnosti, onda možemo da pišemo da je:

$$\lim_{n\to\infty} E_p^1(n) = \frac{S_p^1(n)}{\left\lfloor \frac{n}{2} \right\rfloor} = \frac{n-1}{\left\lfloor \frac{n}{2} \right\rfloor \cdot \log_2 n} = 0$$

U prvom trenutku nam treba $\left\lfloor \frac{n}{2} \right\rfloor = 8$ procesora, ali loše je to što se oni ne koriste u kasnijim koracima. Dakle, ZVRJE nezaposleni !

Analiza slike 2:

Pretpostavimo da su vrednosti $\log_2 n$ i $\frac{n}{\log_2 n}$ celobrojne. Ovu pretpostavku smo uveli radi pojednostavljenja izračunavanja. Podelimo n skalara na $\frac{n}{\log_2 n}$ grupa gde u svakoj grupi ima po $\log_2 n$ elemenata. Svakoj grupi dodelimo po jedan procesor, pa ukupno koristimo $\frac{n}{\log_2 n}$ procesora. Svaki procesor prvo sabere brojeve iz svoje grupe. To sabiranje traje $(\log_2 n)-1$ koraka. Zatim se sabiraju dobijeni rezultati kojih ima $\frac{n}{\log_2 n}$ po istom algoritmu kao na slici 1. To sabiranje se obavi za $\log_2\left(\frac{n}{\log_2 n}\right)$ koraka što je manje od $\log_2 n$.

$$T_1(n) = n-1$$

$$T_1^2(n) = \left((\log_2 n)-1\right) + \log_2\left(\frac{n}{\log_2 n}\right) \approx 2\log_2 n$$

Ubrzanje ovog algoritma je:

$$S_p^2(n) = \frac{T_1(n)}{T_p^2(n)} = \frac{n-1}{2\log_2 n} \approx \frac{1}{2} S_p^1(n)$$

a njegova efikasnost je:

$$E_p^2(n) = \frac{n-1}{\frac{n}{\log_2 n} - 2\log_2 n} = \frac{1}{2}\frac{n-1}{n}$$

$$\lim_{n\to\infty} = E_p^2(n) = \frac{n-1}{\frac{n}{\log_2 n} - 2\log_2 n} = \frac{1}{2}\frac{n-1}{n} = \frac{1}{2}$$

Dakle, u prvom slučaju efikasnost je bila mala kada je problem dosta složen. Ovde kada veličina problema raste, efikasnost ne teži nuli, a ubrzanje je duplo manje nego u prvom slučaju.

Zato što je efikasnost ovde veća algoritam sa slike 2 je bolji.

Zadatak br. 5

Skup naredbi LOAD/STORE mašine je opisan tabelom. Neka 25% ALU operacija koriste neposredno pre učitani operand koji se posle ne koristi. Uvodi se modifikacija ovakve mašine tako što joj se dodaju ALU operacije koje imaju jedan operand u memoriji. Ove operacije imaju 2 taktna ciklusa i povećavaju broj taktnih perioda naredbe grananja te ona sada ima 4 ciklusa takta.

Frekvencija takta se pri tome povćava za 32%.

- Da li su uvedenom modifikacijom poboljšane performanse CPU–a i za koliko su se promenile ?
- Da li se modifikacijom povećava i brzina rada CPU–a ?
- Za koliko procenata se promenio broj MIPS–a koje ima CPU ?

Operacija	Učestanost	Traje taktova
ALU	48%	1
LOAD	21%	2
STORE	12%	2
BRANCH	19%	3

CPI	Srednji broj taktova po instrukciji
CP	Taktni period (clock period)
CPUT	Vreme za koje se koristi CPU (CPU Time)
IC	Broj instrukcija (Instruction Count)

Prvo ćemo razmatrati slučaj pre modifikacije:

$$CPI_0 = \frac{0.48 \cdot 1 \cdot IC_0 + 0.21 \cdot 2 \cdot IC_0 + 0.12 \cdot 2 \cdot IC_0 + 0.19 \cdot 3 \cdot IC_0}{IC_0} = 1.71$$

$$CPUT_0 = IC_0 \cdot CPI_0 \cdot CP_0 = 1.71 \cdot IC_0 \cdot CP_0$$

Sada možemo početi sa analizom sistema posle uvođenja modifikacije.

$$CPI_n = \frac{(1-0.25) \cdot 0.48 \cdot 1 \cdot IC_0 + 0.25 \cdot 0.48 \cdot 2 \cdot IC_0 + (0.21 - 0.25 \cdot 0.48) \cdot 2 \cdot IC_0 + 0.12 \cdot 2 \cdot IC_0 + 0.19 \cdot 4 \cdot IC_0}{IC_n}$$

$$CPI_n = 1.78 \cdot \frac{IC_0}{IC_n}$$

$$CPUT_n = IC_n \cdot CPI_n \cdot CP_n = IC_n \cdot 1.78 \cdot \frac{IC_0}{IC_n} \cdot \frac{CP_0}{1.32} = 1.3485 \cdot IC_0 \cdot CP_0$$

pošto je $CPUT_n < CPUT_0$ zaključujemo da je nova arhitektura bolja. Još ostaje da vidimo za koliko procenata?

$$\frac{CPUT_n}{CPUT_0} = 1 - \frac{n}{100} \Rightarrow n = \frac{CPUT_0 - CPUT_n}{CPUT_0} \cdot 100 = 21.14$$

Dakle, uvedenom modifikacijom postignuto je poboljšanje od oko 21.14%.

$$MIPS_0 = \frac{IC_0}{CPUT_0 \cdot 10^6} = \frac{IC_0}{1.71 \cdot IC_0 \cdot CP_0 \cdot 10^6} = 0.5848 \cdot \frac{1}{CP_0} \cdot 10^{-6}$$

$$MIPS_n = \frac{IC_n}{CPUT_n \cdot 10^6} = \frac{IC_0(1 - 0.48 \cdot 0.25)}{1.3485 \cdot IC_0 \cdot CP_0 \cdot 10^6} = 0.6526 \cdot \frac{1}{CP_0} \cdot 10^{-6}$$

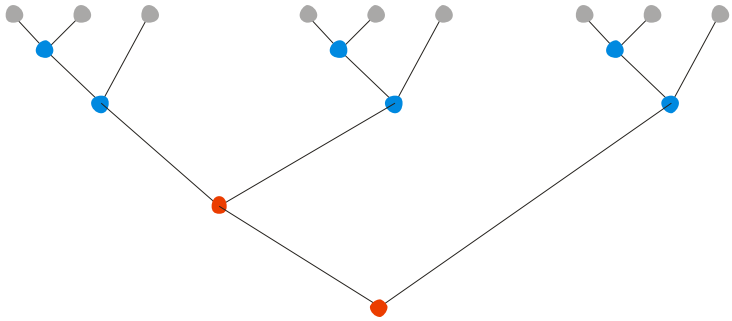
Još ostaje da vidimo koliko je u procentima povećanje broja MIPS–ova.

$$\frac{MIPS_n}{MIPS_0} = 1 + \frac{m}{100} \Rightarrow m = \frac{MIPS_n - MIPS_0}{MIPS_0} \cdot 100 = 11.58\%$$

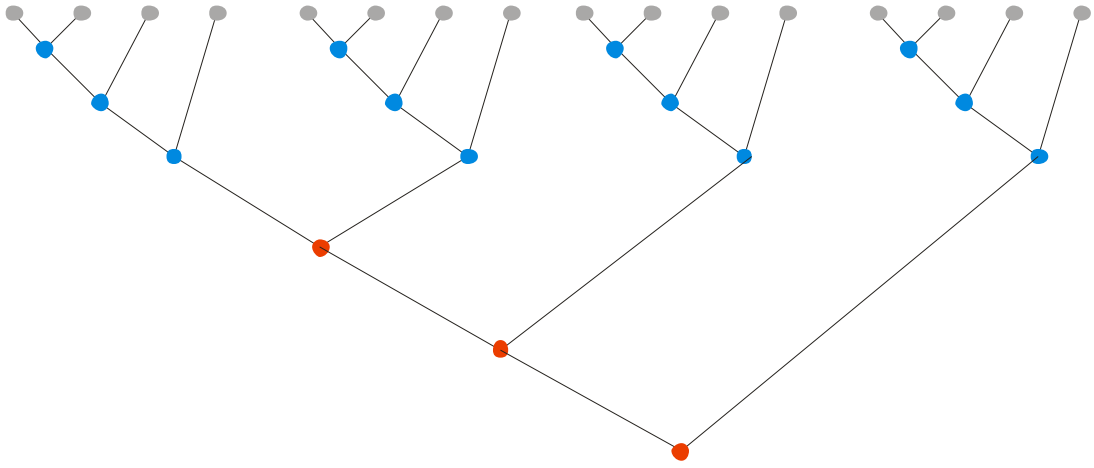
Zadatak br. 6

U višeprocesorskom polju arhitekture tipa linearno polje vrši se sabiranje elemenata matrice dimenzija mxn, pri čemu su m i n kvadrati celih brojeva. Neposredno pre početka sabiranja u lokalne memorije svakog od procesora smeštaju se elementi po jedne vrste ili kolone matrice. Algoritam sabiranja se sastoji u tome što svaki procesor sabira elemente dodeljene vrste, a zatim na dobijenim rezultatima svih procesora primenjuje se sabiranje na sledeći način (za slučaj kada je broj korišćenih procesora 9 odnosno 16).

Slika 1



Slika 2



Odrediti ubrzanje S_p=m i S_p=n i efikasnost E_p=m i E_p=n opisanog algoritma. Odrediti optimalan broj procesora P_s na osnovu razmatranja ubrzanja i optimalni broj procesora na osnovu efikasnosti!

Za oba slučaja izračunati efikasnost algoritma za sabiranje matrice 144x81.

Resenje zadatka br. 6

T_1 je broj koraka za sabiranje mxn elemenata na jednoprocesorskom sistemu

T_p je broj koraka za sabiranje mxn elemenata na predloženom višeprocesoraskom sistemu.

Paralelni računarski sistemi (Elektronski Fakultet – NIŠ)

Marko Miličić

$T_1 = m*n - 1;$

$T_p = T^{I}_{p} + T^{II}_{p} + T^{III}_{p};$

Za slučaj da je p=m;

$$T'_m = n - 1$$

$$T''_m = \frac{m}{\sqrt{m}} - 1 = \sqrt{m} - 1 \qquad S_m = \frac{T_1}{T_m} = \frac{m \cdot n - 1}{n + 2\sqrt{m} - 3}$$

$$T'''_M = \sqrt{m} - 1 \qquad E_m = \frac{m \cdot n - 1}{m\left(n + 2 \cdot \sqrt{m} - 3\right)}$$

$$T_m = n + 2 \cdot \sqrt{m} - 3$$

Za slučaj da je p=n;

$$T'_n = m - 1$$

$$T''_n = \frac{n}{\sqrt{n}} - 1 = \sqrt{n} - 1 \qquad S_n = \frac{T_1}{T_n} = \frac{n \cdot m - 1}{m + 2\sqrt{n} - 3}$$

$$T'''_n = \sqrt{n} - 1 \qquad E_n = \frac{n \cdot m - 1}{n\left(m + 2 \cdot \sqrt{n} - 3\right)}$$

$$T_n = m + 2 \cdot \sqrt{n} - 3$$

Ako je $S_m - S_n > 0$ onda je $P_s = m$

Ako je $S_m - S_n < 0$ onda je $P_s = n$

Što se tiče drugog kriterijuma (efikasnosti) iskoristićemo relacije!

Ako je $E_m - E_n > 0$ onda je $P_e = m$

Ako je $E_m - E_n < 0$ onda je $P_e = n$

Za konkretan slučaj m=144 i n=81 važi:

$$S_m = \frac{T_1}{T_m} = \frac{m \cdot n - 1}{n + 2\sqrt{m} - 3} = \frac{144 \cdot 81 - 1}{81 + 2 \cdot 12 - 3} = \frac{11663}{102} = 114.34$$

$$E_m = \frac{m \cdot n - 1}{m\left(n + 2 \cdot \sqrt{m} - 3\right)} = \frac{144 \cdot 81 - 1}{144 \cdot \left(81 + 2 \cdot 12 - 3\right)} = \frac{11663}{14688} = 0.79$$

$$S_n = \frac{T_1}{T_n} = \frac{n \cdot m - 1}{m + 2\sqrt{n} - 3} = \frac{81 \cdot 144 - 1}{144 + 2 \cdot 9 - 3} = \frac{11663}{159} = 73.35$$

$$E_n = \frac{n \cdot m - 1}{n\left(m + 2 \cdot \sqrt{n} - 3\right)} = \frac{81 \cdot 144 - 1}{81 \cdot \left(144 + 2 \cdot 9 - 3\right)} = \frac{11663}{12879} = 0.90$$

Dakle, za konkretan primer ako izaberemo kriterijum efikasnost biraćemo varijantu P_s=m a ukoliko se odlučimo za kriterijum efikasnosti biraćemo varijantu P_e=n.

Analiza problema je završena !!

PROTOČNA OBRADA

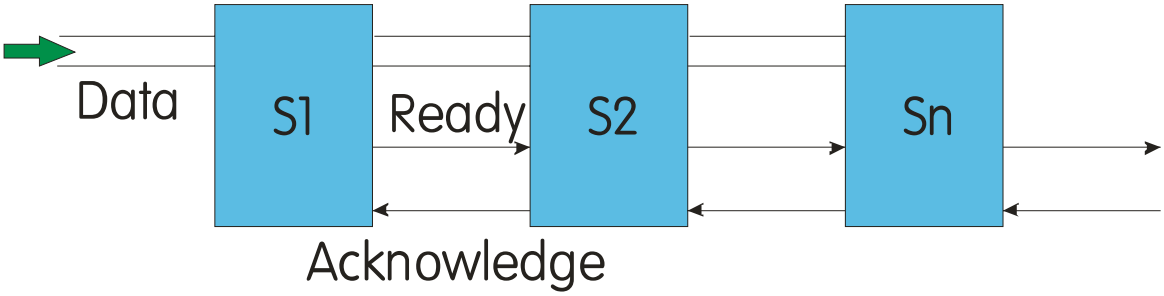
Osnovna ideja je da se ubrza izračunavanje nekih funkcija koje se često koriste u sistemu. To možemo postići **umnožavanjem arhitekture** ili uvođenjem **protočnosti**. Protočnost je način projektovanja hardvera koji se uvodi sa ciljem da se ubrza izračunavanje nekih funkcija. Konkurentnost se uvodi tako što se funkcija razbije na nekoliko podfunkcija pri čemu moraju biti ispunjeni neki uslovi koje moraju da zadovoljavaju te podfunkcije.

Na primer, Neka se funkcija ubrzava tako što se ona razbije na niz podfunkcija f_1 f_2 ...f_k pri čemu, da bi sve teklo bez problema, moraju biti zadovoljeni sledeći uslovi:

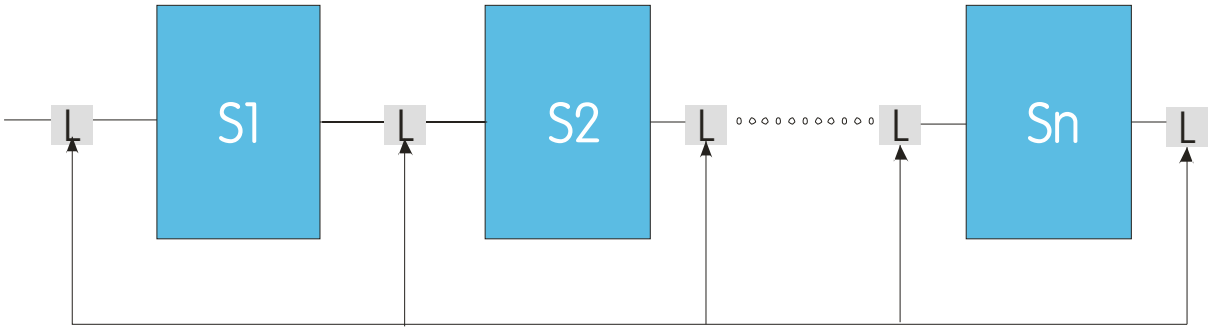
1. Sekvencijalno izvršenje ovih podfunkcija mora da bude ekvivalentno izvršavanju osnovne funkcije.
2. Rezultat jedne podfunkcije predstavlja ulaznu vrednost za drugu podfunkciju koja tek treba da se reši. $f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow ..... \rightarrow f_n$
3. Između ovih podfunkcija osim pomenute zavisnosti ne sme biti bilo kakve dodatne razmene podataka. Dakle, jedina zavisnost između funkcija je ona iz tačke 2.
4. Za svaku od podfunkcija koja treba da se obavi treba da je moguće projektovati hardvare.
5. Vemena potrebna tim hardverima da obave svoju funkciju treba da budu približno jednaka.

Hardverske jedinice koje obavljaju ove podfunkcije nazivaju se **protočni stepeni**. Na osnovu toga kako se obavlja razmena podataka između stepena razlikujemo **sinhrone** i **asinhrone** protočne sisteme.

Kod asinhronih se razmena podataka između stepena upravlja nekom **handshake** procedurom. Kada hardverski stepen završi obradu on obaveštava sledeći stepen preko *Ready* linije da je spreman da mu preda podatak; a ovaj kada je spreman da primi podatak šalje o tome preko *Acknowledge* linije. Posle toga se preko *data* linija vrši prenos podataka. U tom slučaju svaki stepen mora imati memorijske elemente (bafere) za pamćenje podataka. Poželjno je da im je brzina slična ali to kod njih nije ključno.



Kod sinhronog protočnog sistema postoji jedinstveni clock (takt) signal koji tačno definiše trenutke u kojima se vrši razmena podataka između stepena. Ovde protočni stepeni ne sadrže memorijske elemente već se između stepena ubacuju lečevi (najčešće realizovani pomoću MS FlipFlopova), a razmenom podataka se upavlja tako što se isti klok signal dovede do svih lečeva istovremeno. Razmena podataka vrši se u trenutku nailaska klok signala.



Svaki od stepena unosi svoje kašnjenje ali i lečevi unose svoje kašnjenje. Perioda kloka (takta) se određuje na osnovu dužine obrade u stepenu koji je najsporiji (zahteva najviše vremena) Upravo zbog toga se pri projektovanju protočnih sistema zahteva da obrada podataka u pojedinim protočnim stepenima bude približno ista za sve procesne elemente, da neki sporiji elementi ne bi usporavali one brže. Dakle:

$$T = \max\{\tau_1, \tau_2, ...\tau_k\} + \tau_L$$

Zbog konačnog vremena prostiranja signala kroz provodnik dolazi do tzv. **krivljenja signala**, pa je potrebno odrediti najmanju moguću širinu impulsa signala kloka. Naime, ovaj impuls mora da traje duže od najvećeg mogućeg propagacionog kašnjenja kroz provodnik.

Primer:

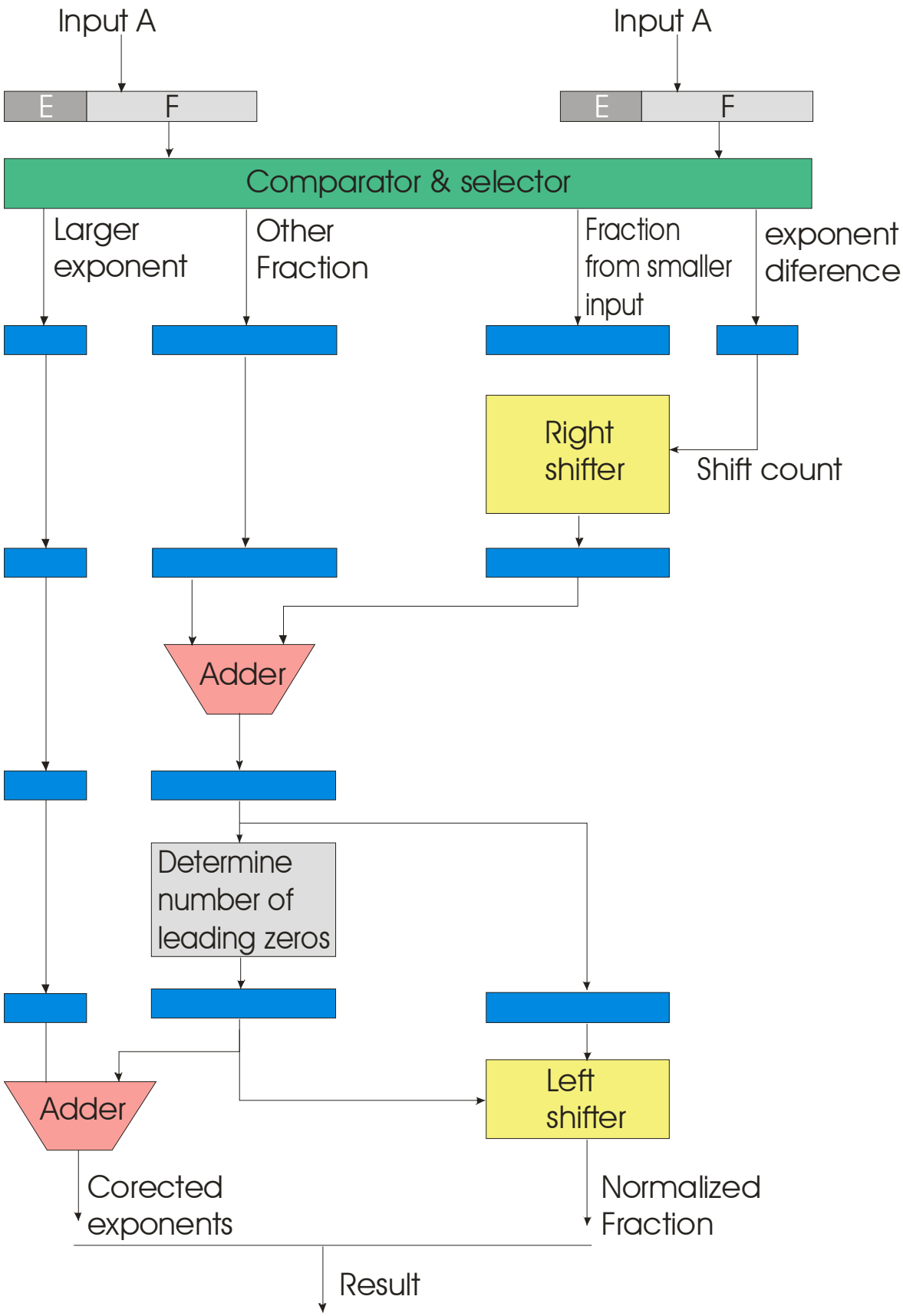
Sabirač za sabiranje dva broja u pokretnom zarezu:

$A=a * 2^p$ $B=b * 2^q$

1. Poređenje eksponenata p i q i nalaženje većeg od njih, kao i njihove razlike. V=max(p,q) t=(p–q).
2. Pomeranje mantise manjeg broja za t mesta u desno.
3. Sabiranje mantisa
4. Određujemo broj važećih nula u sumi U
5. Pomeramo mantisu broja za U mesta ulevo i ažuriramo eksponent S=r+u

Jako je bitno da je moguće projektovati hardver za svaku od ovih faza.

Primer ovakvog sabirača prikazan je na sledećoj slici:


Input A
Input A
E
F
E
F
Comparator & selector
Larger exponent
Other Fraction
Fraction from smaller input
exponent diference
Right shifter
Shift count
Adder
Determine number of leading zeros
Adder
Left shifter
Corected exponents
Normalized Fraction
Result

Ukoliko pretpostavimo da su latencije ovih stepena u protočnoj obradi sledeće:

$$\tau_1 = 60ns$$
$$\tau_2 = 50ns$$
$$\tau_3 = 80ns$$
$$\tau_4 = 50ns$$
$$\tau_5 = 80ns$$
$$\tau_L = 10ns$$

T=80ns+10ns=90ns

Tnp=320 ns Tnp se odnosi na neprotočni sabirač

Tp=90*5=450ns tp se odnosi na protočni sabirač

				P_1	P_2	P_3	P_4	P_5	S_5
			P_1	P_2	P_3	P_4	P_5		S_4
		P_1	P_2	P_3	P_4	P_5			S_3
	P_1	P_2	P_3	P_4	P_5				S_2
P_1	P_2	P_3	P_4	P_5					S_1

Nakon inicijalnog vremena od 450ns sledeći rezultat se dobija nakon samo 90ns. To je vreme koje se zove **latentnost procesora**. Ili drugim rečima, to je vremo koje je potrebno da se svi protočni stepeni napune.

Tnp=n*320ns = 3.6

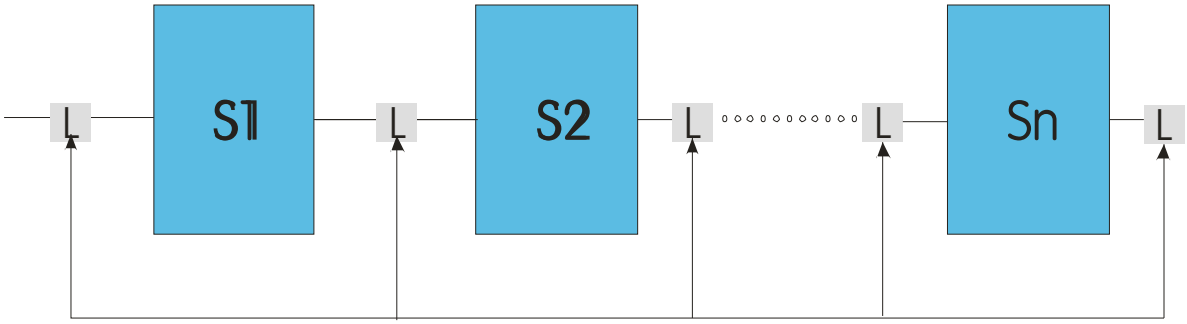
Tp=450+(n–1)*90 = 1.36

Dakle, protočni sistem je sporiji ukoliko se koristi za mali obim posla, odnosno za mali broj izračunavanja (manji ili jednak od onog potrebnog da se svi stepeni napune), ali je zato znatno brži kada se radi o velikom obimu posla jer posle proteklog vremena da se napuni protočni sistem, rezultate dobijamo mnogo brže nego kada ne bi koristili protočni sistem već običan, sekvencionalni.

Podela protočnih sistema

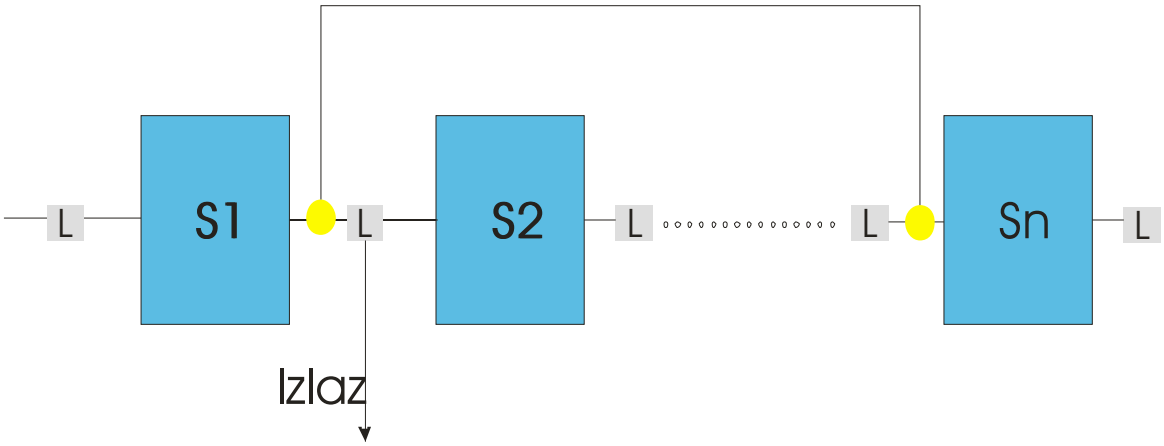
U odnosu na način sprezanja hardverskih stepena protočni sistemi se dele na:

1. Linearni protočni sistemi (postoje kaskadne veze)
2. Nelinearni protočni sistemi (veze mogu biti kaskadne, povratne, izvedene unapred ili unazad)



Kod linearnih sistema postoje samo kaskadne veze među stepenima, dok kod nelinearnih sistema postoje i povratne veze. Ovo je slika tipičnog linearnog protočnog sistema, dok će sledeća slika predstavljati nelinearni protočni sistem.

Sledeća slika se od ove razlikuje upravo po tome što osim kaskadnih veza postje i povratne veze tako da je sledeća slika tipičan primer nelinearnog protočnog sistema.



Važno je primetiti da kod nelinearnog protočnog sistema postoje i povratne veze kojih kod kaskadnog (linearnog) protočnog sistma nema.

Naravno, protočni sistemi se pored svoje linearnosti dele i po mogućnosti obrade na:

1. Jednofunkcijske (obavljaju istu operaciju za svaki ulazni podatak)
2. Višefunkcijske (mogu obavljati različite obrade u istom ili različitim vremenskim trenucima, i upravo zato su nam potrebne kaskadne i povratne veze).

Naravno, nikad kraja podelama. Višefunkcijski sistemi se pak dele na:

1. Statički konfigurisane
2. Dinamički konfigurisane

Statički konfigurisani su sistemi kod kojih se u datom trenutku može izvršavati samo jedna f–ja i između 2 konfiguracije takav sistem se ponaša kao jednofunkcijski protočni sistem. Ako se često menja tip obrade performanse sistema su loše.

Dinamički konfigurisani sistemi su sistemi kod kojih se tip obrade može menjati za svaki ulazni podatak. U istom vremenskom trenutku je moguće izvršenje više f–ja pod uslovom da ne koristi iste hardverske resurse (hardverske stepene). Sa stanovišta performansi ovi protočni sistemi su bolji ali i skuplji i teži za pravljenje. Takođe je i upravljanje složenije. Dakle, ukoliko nam je ekonomski faktor od presudnog značaja odlučićemo se za statički konfigurisane protočne sisteme a ukoliko nam je kvalitet na prvom mestu odlučićemo se za dinamički konfigurisane protočne sisteme.

ZADACI (PROTOČNA OBRADA)

Rekapitulacija teorije

Postoji više načina za klasifikovanje paralelnih izračunavanja, a ta klasifikacija se može izvršiti na osnovu:

1. **Načina na koji je izračunavanje izdeljeno**

- Deljenje Paralelno po podacima
- Deljenje paralelno po funkciji

2. **Načina na koji se izračunavanje izvršava**

- Protočno izvršavanje
- Konkurentno izvršavanje

Prvi kriterijum:

Što se tiče prvog kriterijuma i to, izvršavanja parlelno po podacima podrazumeva da je skup podataka nad kojim se vrši izraunavanje raspodeljen po procesorima. Procesori taba izvršavaju isti program, ali nad različitim skupom podataka.

Ukoliko se radi o paralelnom deljenju po funkciji, onda se vrši dekompozicija programa na module koji izvršavaju različite funkcije i koji se mogu izvršavati na više procesora.

Drugi kriterijum:

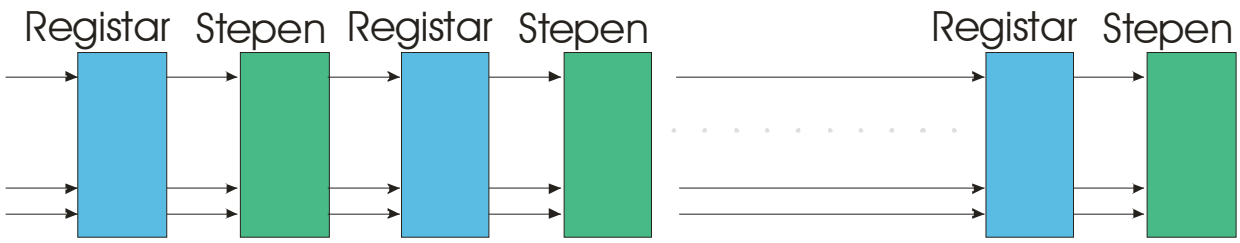
Konkurentno izvršavanje koristi prostorni paralelizam, tako što više procesora izvršavaju više nezavisnih task–ova (zadataka).

Protočno izračunavanje koristi vremenski paralelizam i ovde podaci teku kroz sistem a komunikacja se izvršava među susednim elementima (procesorima) tako da će podatak koji jednom uđe u sistem biti više puta korišćen.

Naravno, postoje i kombinacije ovih izračunavanja, kojim dobijamo već dobro poznatu klasifikaciju na:

- MIMD računare
- SIMD računare
- MISD računare
- SISD računare

Parametri i performanse protočnih sistema



Registri su sastavljni od flipflop–ova i kontrolisani su zajedničkim clock signalom (svaki stepen je kombinaciona mreša koja obavlja neku funkciju nad ulaznim podacima). Broj flip flop–ova zavisi od količine podataka koji se kreću kroz sistem.

Važni parametri protočnog sistema su:

N	broj stepena u sistemu
N_R	ukupan broj flip–flop–ova u registrima
K_R	Ukupna cena koštanja N stepena u jedinici vremena
T_S	Maksimalno propagaciono kašnjenje kroz jedan stepen
T_R	maksimalno propagaciono kašnjenje po registru
M	Broj operacija za čije izvršenje se koristi protočni stepen
K_R	cena koštanja jednog flip–flopa u jedinici vremena

Maksimalno propagaciono kašnjenje se uzima da bi za jednu periodu kloka (T_S+T_R) svi podaci stigli da prođu kroz odgovarajući registar ili stepen.

Performanse protočnih sistema se karakterišu specijalnim veličinama:

$\tau(M)$	Vreme potrebno za izvršenje jedne operacije kada se izvršava M operacija
$\mu(M)$	Cena jedne operacije kada se izvršava M operacija
δ	kašnjenje sistema, odnosno, vreme koje protekne od unošenja prvog unosa u sistem i startovanja prve operacije do dobijanja prvog rezultata na izlazu sistema.

Za sistem na slici možemo da napišemo:

$$\delta = N \cdot (T_S + T_R)$$

$$\tau(M) = \left[N \cdot (T_S + T_R) + (M-1) \cdot (T_S + T_R)\right] \cdot \frac{1}{M} = (T_S + T_R) \cdot \frac{M+N-1}{M}$$

$$\tau = \lim_{M \to \infty} \tau(M) = T_S + T_R$$

$$\mu(M) = (K_P + N_R K_R) \cdot \tau(M) = (K_P + N_R K_R) \cdot (T_S + T_R) \cdot \frac{M + N - 1}{M}$$

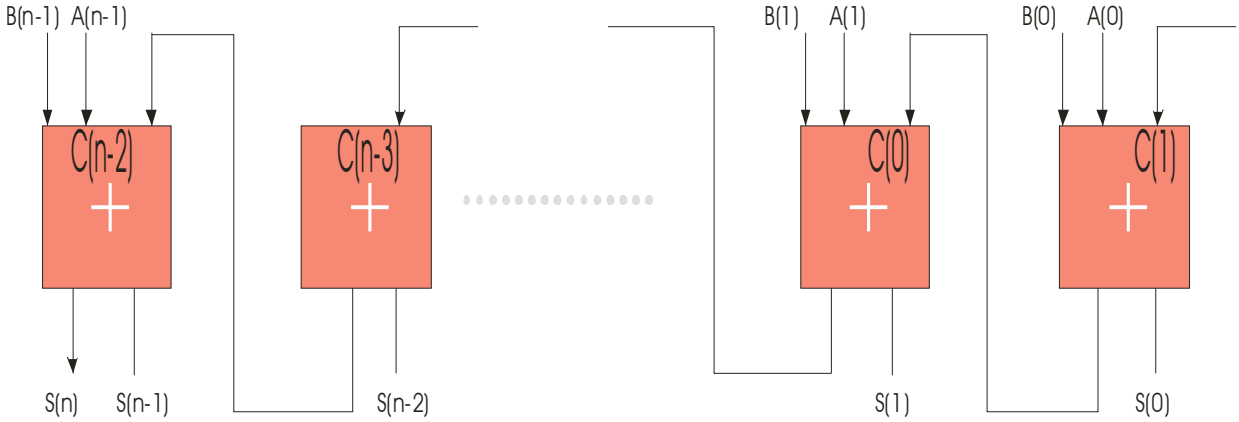
$$\mu = \lim_{M \to \infty} \mu(M) = (K_P + N_R K_R) \cdot (T_S + T_R)$$

Zadatak br. 1

1. Nacrtati šemu paralelnog sabirača n–to bitnih brojeva.
2. Na osnovu rešenja pod 1. projektovati protočni sabirač
3. Za sabirače dobijene pod 1. i 2. odrediti sve parametre performansi i uporediti ih.

Neka su cena leča C_L=3, kašnjenje kroz leč T_L=2, cena sabirača C_S=16, kašnjenje kroz sabirač T_S=5

Rešenje zadatka br. 1



Dakle, sa C je označen prenos iz predhodnog stepena. U prvom sabiraču, naravno, prenosa nema jer ispered njega nema ničega, te nema ko da mu prosledi bit prenosa. Ovaj bit prenosa prenosi se lančano kroz sabirače, sve do poslednjeg sabirač. Sa S su označeni zbirovi. Tj. u prvom sabiraču se na liniji zbira dobija zbir prvih bitova brojeva a i b, u drugom se dobija zbir drugih bitova i tako redom. Naravno, ne treba zaboraviti ni bitove prenosa koji se takođe sabiraju sa izvornim bitovima (bitovima operanada).

Sada možemo odrediti kašnjenje sistema. Samo da podsetimo šta to beše kašnjenje sistema!

Kašnjenje sistema je vreme koje protekne od unošenja prvog unosa u sistem i startovanja prve operacije do dobijanja prvog rezultata na izlazu sistema.

$$\delta = N \cdot (T_S)$$

U ovom izrazu T_S predstavlja maksimalno propagaciono kašnjenje u jednom stepenu tj. u našem slučaju to je jedan sabirač.

Dakle, sasvim je logično da je kašnjenje sistema jednako proizvodu kašnjenja kroz jedan stepen i broja stepeni.

Sledeća mera performansi je **vreme potrebno za izvršenje jedne operacije kada se izvršava M operacija**:

$$\tau(M) = \frac{M \cdot N \cdot T_S}{M} = N \cdot T_S$$

Pošto u našem modelu nema protočnosti, uopšte nismo morali ništa da računamo. Mogli smo odmah da zaključimo da je to vreme jednako ukupnom kašnjenju sistema. Međutim, da bi malo bolje razjasnili ovu meru performansi prodiskutovaćemo ovaj izraz.

Dakle, ukoliko nama treba vreme koje je potrebno za izvršavanje jedne naredbe, a ukupan posao je izračunavanje M naredbi, onda ćemo ukupno kašnjenje sistema pomnožiti sa M i dobiti koliko je

vremena potrebno za izračunavanje M naredbi. Tako dobijeno vreme možemo podeliti sa M da bi smo dobili vreme potrebno za izvršavanje jedne operacije iz niza od 1 do M. Dakle, prvo smo množili sa M pa onda delimo sa M. Zaključujemo da nam opet ostaje samo ukupno kašnjenje kroz sistem.

Poželjno je znati graničnu vrednost ove performansne mere. Odnosno, vrednost kojoj teži potrebno vreme, kada broj naredbi koje se trebaju obaviti teže beskonačnosti. To je slučaj kada red veličine problema teži beskonačnosti. Sledeći izraz upravo govori o tome.

$$\tau = \lim_{M \to \infty} \tau(M) = N \cdot T_S$$

Sledeća performansna mera je takozvana **cena jedne operacije kada se obavlja M operacija**. Ova mera je vrlo slična malopre opisanoj performansnoj meri. Dakle, ukoliko sa N (ponovo) označimo broj stepeni protočnog sistema, onda je proizvod N i C_S jednak ukupnoj ceni za ceo sistem koji se sastoji od N stepeni. Ukoliko cenu kašnjenja celog sistema pomnožimo sa malopre dobijenom vrednosti za vrme kašnjenja potrebno za izvršavanje jedne instrukcije dobićemo:

$$\mu(M) = N \cdot C_S \cdot \tau(M) = N^2 \cdot T_S \cdot C_S$$

Dakle, generalni zaključak će biti da je od interesa obično baš taj proizvod i da se najčešće upravo on traži, kao najobjektivnija performansna mera. Dakle, da ponovimo još jedared:

Proizvod cene kašnjenja celog sistema sa kašnjenjem potrebnim da se izvrši jedna instrukcija dobijamo performansku meru koju nazivamo **CENA JEDNE OPERACIJE**.

$$\mu = \lim_{M \to \infty} \mu(M) = N^2 \cdot T_S \cdot C_S$$

Naravno, od interesa je poznavati i graničnu vrednost ove veličine, jer u realnom problemu obično boroj operacija teži beskonačnosti, tako da nije na odmet da pomenemo i ovu graničnu vrednost.

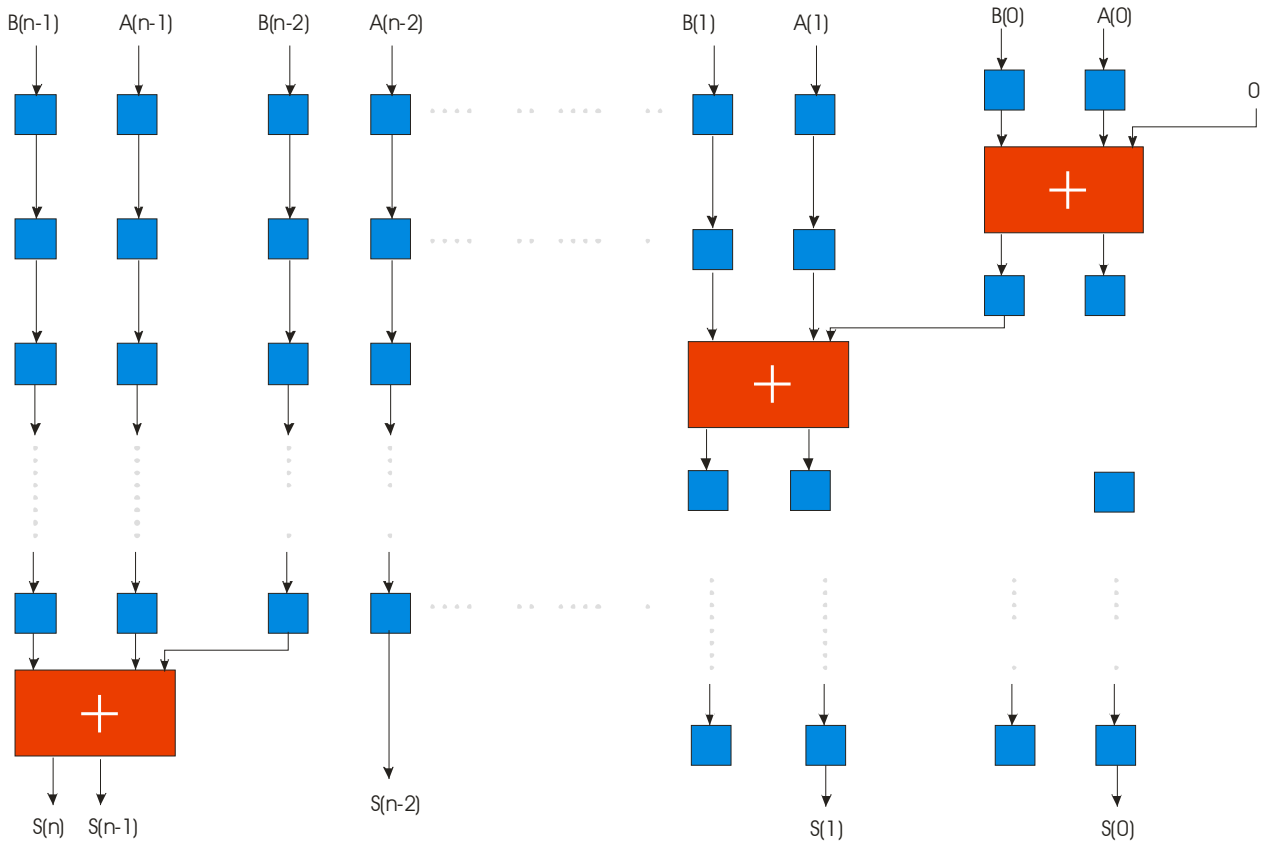
Protočni sabirač:

Važno je primetiti dijagonalnu simetriju tj. treba primetiti da su sabirači postavljeni po dijagonali. Postavlja se pitanje zašto je to tako ?

Hajde zajedno da razmislimo!

Ukoliko bi svi stepeni bili u jednom redu ove strukture koja nas podseća na matricu, da li bi onda bili u mogućnosti da preuzimamo podatke iz lečeva prethodnog stepena? Narvno da da. Znači, nije u tome poblem. Problem je u tome što ukoliko ne bi bili postavljeni dijagonalno, onda ne bi moglo puniti protočno sistem, odnosno, ne bi bila moguća protočnost. Sve u svemu, pokušajmo da zmislimo kakav bi haos nastao kada bi u neprotočni sistem pokušali da punimo podatke, kao da je protočan. Sve bi se ispomešalo i na kraju bi kao rezultat dobili nešto što nema veze sa stvarnim rezultatom.

Dakle, ovakva struktura je neminovna!



Sada, kada imamo šemu možemo početi analizu!

Kao i u prošlom primeru, tražimo **ukupno kašnjenje sistema**. To kašnjenje dobijamo kada pomnožimo broj protočnih stepeni sa kašnjenjem jednog stepena. O ovom primeru, se kašnjenje jednog stepena sastoji od kašnjenja samog sabirača i naravno, kašnjenja kroz leč. Ukolikobi smo to hteli da zapišemo dobili bi smo formulu:

$$\delta_P = N \cdot (T_S + T_L)$$

E, što se tiče vremena potrebnog da se obavi jedna operacija, pri obavljanju posla od ukupno M operacija, morali bi malo više da razmislimo. Naime, Prvo ćemo izračunaki koliko je vreme potrebno da se izvrši ceo posao u protočnom sistemu, a zatim ćemo to vreme podeliti sa M da bi dobili vreme potrebno za izračunavanje jede operacije.

Dakle, Vreme potrebno za izvršavanje celog posla se satoji od dve komponente. Te komponente su

- vreme koje je potrebo da se sistem napuni, a ono je jednako ukupnom vremenu kašnjenja kroz sistem $\delta_P = N \cdot (T_S + T_L)$
- vreme koje je potrebno da protekne od tog trenutka do završetka posle.

Ova druga komponenta je malo komplikovanija za analizu, pa i za samo shvatanje. Ali krenimo redom.

Vreme kašnjenja kroz jedan stepen je T_S+T_L zar ne?

Ostaje sada pitanje koliko još instrukcija se treba izvršiti, odnosno, kroz koliko još stepeni se treba proći da bi se dobili rezultati svih M naredbi.

Pošto je jasno za vreme punjenja sistema nije moglo biti obavljeno više od jedne naredbe, onda je jasno da još treba obaviti M–1 operaciju.

Sledeći izraz nam upravo to kazuje.

$$\tau(M)_P = \left[N \cdot (T_S + T_L) + (M-1) \cdot (T_S + T_L)\right] \cdot \frac{1}{M} = (T_S + T_L) \cdot \frac{M+N-1}{M}$$

Naravno, interesuje nas i granična vrednost

$$\tau_P = \lim_{M \to \infty} \tau(M) = T_S + T_L$$

Sada bi trebali da izračunamo ukupan broj lečeva.

Na prvom nivou ima 2*N lečeva, na drugom takođe 2*N, na trećem 2(N–1)+1, na četvrtom 2(N–2)+2 i na poslednjem 2*2+(n–2). Sada možemo izračunati ukupan broj lečeva:

$$= 2n + 2n + \left[2(n-2)+1\right] + \left[2(n-2)+2\right] + \ldots\ldots\ldots + \left[2 \cdot 2 + (n-2)\right] =$$

Ovaj izraz treba malo srediti i na kraju će se dobiti: (treba malo matematike i razmišljanja)

$$= 3(1 + 2 + 3 + \ldots\ldots\ldots + n) - 1 = 3 \cdot \frac{n(n+1)}{2} - 1 = \frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1$$

Neko se možda zapitao zašto računamo uopšte ukupan broj lečeva?

Zato što Unupnu cenu kašnjenja celog sistema dobijamo tako što saberemo ukupnu ceu kašnjenja koja potiče od svih sabirača i ukupnu cenu kašnjenja koja potiče od SVIH lečeva u sistemu. Tako dobijenu cenu kašnjenja pomnožimo sa vremenom kašnjenja i dobijemo:

$$\mu(M) = \left(N \cdot C_S + \left(\frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1\right) \cdot C_L\right) \tau(M) = \left(N \cdot C_S + \left(\frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1\right) \cdot C_L\right) \cdot (T_S + T_L) \cdot \frac{M+N-1}{M}$$

kao i obično granična vrednost je:

$$\mu = \lim_{M \to \infty} \mu(M) = \left(N \cdot C_S + \left(\frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1\right) \cdot C_L\right) \cdot (T_S + T_L)$$

Setimo se da su vrednosti: C_L=3, T_L=2, C_S=16, T_S=5

Sada možemo napraviti paralelno poređenje između protočnog i neprotočnog sistema.

Za neprotočni sistem važi:

$$\delta = N \cdot (T_S) = 5 \cdot N$$

$$\tau(M) = \frac{M \cdot N \cdot T_S}{M} = N \cdot T_S = 5 \cdot N \qquad \tau = \lim_{M \to \infty} \tau(M) = N \cdot T_S = 5 \cdot N$$

$$\mu(M) = N \cdot C_S \cdot \tau(M) = N^2 \cdot T_S \cdot C_S = 80 \cdot N^2 \qquad \mu = \lim_{M \to \infty} \mu(M) = N^2 \cdot T_S \cdot C_S = 80 \cdot N^2$$

Za protočni sistem važi:

$$\delta_P = N \cdot (T_S + T_L) = 7 \cdot N$$

$$\tau(M)_P = [N \cdot (T_S + T_L) + (M-1) \cdot (T_S + T_L)] \cdot \frac{1}{M} = (T_S + T_L) \cdot \frac{M+N-1}{M} = 7 \cdot \frac{M+N-1}{M}$$

$$\tau_P = \lim_{M \to \infty} \tau(M) = T_S + T_L = 7$$

$$\mu_P(M) = \left(N \cdot C_S + \left(\frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1 \right) \cdot C_L \right) \tau(M) = \left(N \cdot C_S + \left(\frac{3}{2} \cdot n^2 + \frac{3}{2} \cdot n - 1 \right) \cdot C_L \right) \cdot (T_S + T_L) \cdot \frac{M+N-1}{M} =$$

$$= \left(\frac{63}{2} \cdot N^2 + \frac{287}{2} N - 21 \right) \frac{M+N-1}{M}$$

$$\mu_P = \lim_{M \to \infty} \mu(M) = \frac{63}{2} \cdot N^2 + \frac{287}{2} N - 21$$

Uzmimo na primer da je N=8 (osmostepeni protočni sistem)

Tada ćemo dobiti:

$$\mu = \lim_{M \to \infty} \mu(M) = N^2 \cdot T_S \cdot C_S = 80 \cdot N^2 = 5120$$

$$\mu_P = \lim_{M \to \infty} \mu(M) = \frac{63}{2} \cdot N^2 + \frac{287}{2} N - 21 = 3143 + \frac{22001}{M}$$

Dakle, ukolko izjednačimo ove dve vrednosti dobićemo jednačinu po M. Kada rešimo to M dobićemo da je ono M=11.12.

Dakle, možemo da zaključimo da će protočni sabirač biti bolji od neprotočnog tek kada je M>11 tj. za minimum 12 operacija.

Uzmimo na primer da je N=16

Tada ćemo dobiti:

$$\mu = \lim_{M \to \infty} \mu(M) = N^2 \cdot T_S \cdot C_S = 80 \cdot N^2 = 20480$$

$$\mu_P = \lim_{M \to \infty} \mu(M) = \frac{63}{2} \cdot N^2 + \frac{287}{2} N - 21 = 10339 + \frac{155085}{M}$$

Dakle, ukolko izjednačimo ove dve vrednosti dobićemo jednačinu po M. Kada rešimo to M dobićemo da je ono M=15.29.

Dakle, možemo da zaključimo da će protočni sabirač biti bolji od neprotočnog tek kada je M>15 tj. za minimum 16 operacija.

Zadatak br. 2

1. Grafički predstaviti tok podataka pri množenju 4 bitnih neoznačenih brojeva na protočnom Milerovom množaču.

Neka je:

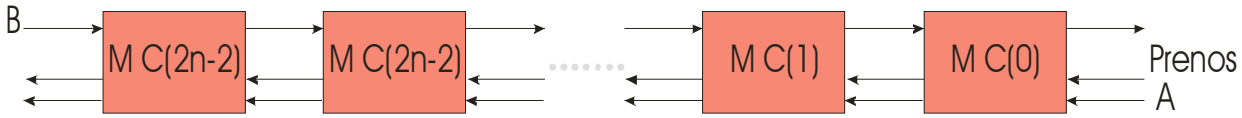
Cena leča $C_L=3$, kašnjenje kroz leč $T_l=2$

Cena sabirača $C_S=16$, kašnjenje kroz sabirač $T_S=5$

Cena AND koloa je $C_{AND}=3$, kašnjenje kroz AND kolo je $T_{AND}=2$.

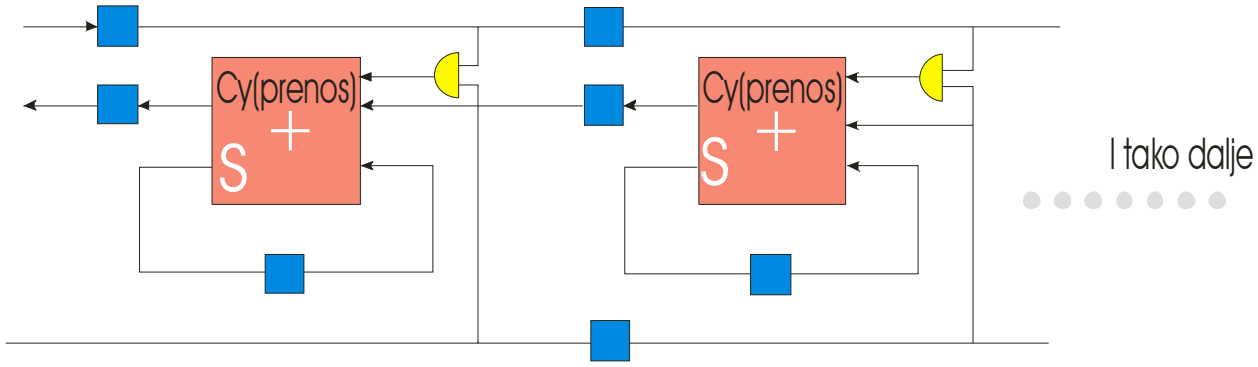
2. Kakve su performanse ovakvog množača ako se obavlja M uzastopnih operacija $C_{i+1}=C_i+A_iB_i$, i=1,2,3... gde je početni element niza $C_0=0$.

Rešenje zadatka br. 2



MC je oznaka za Milerovu ćeliju

Sada treba sastaviti šemu celokupnog Milerovog brojača.



$T_C=T_S+T_{AND}+T_L$

Sledeća tablica prikazuje kako se množe dva 4–bitna broja

A_3	A_2	A_1	A_0	X	B_3	B_2	B_1	B_0	=	A_3B_0	A_2B_0	A_1B_0	A_0B_0
									A_3B_1	A_2B_1	A_2B_1	A_0B_1	
								A_3B_2	A_2B_2	A_1B_2	A_0B_2		
							A_3B_3	A_2B_3	A_1B_3	A_0B_3			
						P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

A ova tabela prikazuje tok podataka u Milerovom brojaču.

	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	
B_0->t_1	$B_0$0	0	0	0	0	0	0	0	<-0
0->t_2	0	$B_0$0	0	0	0	0	0	0A_3	<-A_3
B_1->t_3	$B_1$0	0	$B_0$0	0	0	0	0A_3	0	<-0
0->t_4	0	$B_1$0	0	$B_0$0	0	0A_3	0	0A_2	<-A_2
B_2->t_5	$B_2$0	0	$B_1$0	0	B_0A_3	0	0A_2	0	<-0
0->t_6	0	$B_2$0	0	B_1A_3	0	B_0A_2	0	0	<-A_1
B_3->t_7	$B_3$0	0	B_2A_3	0	B_1A_2	0	B_0A_1	0	<-0
0->t_8	0	B_3A_3	0	B_2A_2	0	B_1A_1	0	B_0A_0	<-A_0
0->t_9	0A_3	0	B_3A_2	0	B_2A_1	0	B_1A_0	0	<-0
0->t_{10}	0	0A_2	0	B_3A_3	0	B_2A_0	0	$B_1$0	<-0
0->t_{11}	0A_2	0	0A_1	0	B_3A_0	0	$B_2$0	0	<-0
0->t_{12}	0	0A_1	0	0A_0	0	$B_3$0	0	$B_2$0	<-0
0->t_{13}	0A_1	0	0A_0	0	0	0	$B_3$0	0	<-0
0->t_{14}	0	0A_0	0	0	0	0	0	$B_3$0	<-0
0->t_{15}	0A_0	0	0	0	0	0	0	0	<-0
0->t_{16}	0	0	0	0	0	0	0	0	<-0

$$\mu = \lim_{M\to\infty} \mu(M) = 24552$$

U ljubičastim kvadratićima se nalaze proizvodi.

U trenutku t_{12} može ući novi operand (jer će se tako sresti u trenutku t_{16} pa se neće narušiti množenje. Ovo je jako bitno zapažanje.

$T_c=T_L+T_S+T_{AND}=2+2+5=9$

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(3n-1)T_C}{M}\Big|_{n=4} = 99 + \frac{36}{M}$$

PITAM SE DA II JE 3N-1 ILI 3N ????

$$\tau = \lim_{M\to\infty} \tau(M) = 99$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=4} = 24552 + \frac{8928}{M}$$

Prvi deo zadatka smo uradili, ali sada ostaje problem, kako izračunati performanse ukoliko se obavlja M uzastopnih operacija $C_{i+1}=C_i+A_iB_i$

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(2n-1)T_C}{M}\Big|_{n=4} = 63 + \frac{72}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 63$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\big|_{n=4} = 15624 + \frac{17856}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 15624$$

Zadatak br. 3

Grafički predstaviti tok podataka kod množenja 7–bitnih neoznačenih celih brojeva na protočnom Milerovom množaču. Kakve su performanse ovog množača ako se obavlja M uzastopnih množenja bez akumulacije, a kakve sa akumulacijom.

Cena leča je C_L=2

Cena sabirača je C_S=15

Cena AND kola je C_{AND}=5

Kašnjenja su:

Kašnjenje leča T_L=3

Kašnjenje sabirača T_S=7

Kašnjenje AND kola je T_{AND}=4

Rešenje zadatka br. 3

a) bez akumulacije:

$T_C = T_L + T_{AND} + T_{ADD} = 14$

$C = (4N - 1) * T_C$

Za N = 7 (sedmostepeni protočni sistem) se dobija:

C = 27 * 14 = 378

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(3n)T_C}{M}\Big|_{n=7} = 924 + \frac{84}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 294$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\big|_{n=7} = 392 \cdot \tau(M) = 115248 + \frac{928}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 115248$$

a) sa akumulacijom:

$T_C = T_L + T_{AND} + T_{ADD} = 14$

$C = (4N - 1) * T_C$

Za N = 7 (sedmostepeni protočni sistem) se dobija:

C = 27 * 14 = 378

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(2n-1)T_C}{M}\Big|_{n=7} = 182 + \frac{196}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 182$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=7} = 71844 + \frac{76832}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 71844$$

Grafički prikaz toka podataka dat je sa:

	P13	P12	P11	P10	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0	
B_0	$B_0$0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	$B_0$0	0	0	0	0	0	0	0	0	0	0	0	0A_6	A_6
B_1	$B_1$0	0	$B_0$0	0	0	0	0	0	0	0	0	0	0A_6	0	0
0	0	$B_1$0	0	$B_0$0	0	0	0	0	0	0	0	0A_6	0	0A_5	A_5
B_2	$B_2$0	0	$B_1$0	0	$B_0$0	0	0	0	0	0	0A_6	0	0A_5	0	0
0	0	$B_2$0	0	$B_1$0	0	$B_0$0	0	0	0	0A_6	0	0A_5	0	0A_4	A_4
B_3	$B_3$0	0	$B_2$0	0	$B_1$0	0	$B_0$0	0	0A_6	0	0A_5	0	0A_4	0	0
0	0	$B_3$0	0	$B_2$0	0	$B_1$0	0	B_0A_6	0	0A_5	0	0A_4	0	0A_3	A_3
B_4	$B_4$0	0	$B_3$0	0	$B_2$0	0	B_1A_6	0	B_0A_5	0	0A_4	0	0A_3	0	0
0	0	$B_4$0	0	$B_3$0	0	B_2A_6	0	B_1A_5	0	B_0A_4	0	0A_3	0	0A_2	A_2
B_5	$B_5$0	0	$B_4$0	0	B_3A_6	0	B_2A_5	0	B_1A_4	0	B_0A_3	0	0A_2	0	0
0	0	$B_5$0	0	B_4A_6	0	B_3A_5	0	B_2A_4	0	B_1A_3	0	B_0A_2	0	0A_1	A_1
B_6	$B_6$0	0	B_5A_6	0	B_4A_5	0	B_3A_4	0	B_2A_3	0	B_1A_2	0	B_0A_1	0	0
0	0	B_6A_6	0	B_5A_5	0	B_4A_4	0	B_3A_3	0	B_2A_2	0	B_1A_1	0	B_0A_0	A_0

DOVRŠITE SAMI :)

Zadatak br. 4

Grafički predstaviti tok podataka kod množenja 3–bitnih neoznačenih celih brojeva na protočnom Milerovom množaču. Kakve su performanse ovog množača ako se obavlja M uzastopnih množenja bez akumulacije i sa akumulacijom. C_L=2, C_S=15, C_{AND}=5; T_L=3, T_S=7, T_{AND}=4

Rešenje zaradtka br. 4

a) bez akumulacije:

$T_C = T_L + T_{AND} + T_{ADD} = 14$

$C = (4N - 1) * T_C$

Za N = 3 (trostepeni protočni sistem) se dobija:

C = 11 * 14 = 154

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(3n)T_C}{M}\Big|_{n=3} = 112 + \frac{42}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 112$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=3} = 392 \cdot \tau(M) = 18816 + \frac{7056}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 18816$$

a) sa akumulacijom:

$T_C = T_L + T_{AND} + T_{ADD} = 14$

$C = (4N - 1) * T_C$

Za N = 3 (trostepeni protočni sistem) se dobija:

C = 27 * 14 = 378

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(2n-1)T_C}{M}\Big|_{n=3} = 70 + \frac{84}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 70$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=3} = 11760 + \frac{14616}{M}$$

$$\mu = \lim_{M \to \infty} \mu(M) = 11760$$

Tok podataka prikazan je sledećom tabelom:

	P5	P4	P3	P2	P1	P0	
B_0	$B_0 0$	0	0	0	0	0	0
0	0	$B_0 0$	0	0	0	$0 A_2$	A_2
B_1	$B_1 0$	0	$B_0 0$	0	$0 A_2$	0	0
0	0	$B_1 0$	0	$B_0 A_2$	0	$0 A_1$	A_1
B_2	$B_2 0$	0	$B_1 A_2$	0	$B_0 A_1$	0	0
0	0	$B_2 A_2$	0	$B_1 A_1$	0	$B_0 A_0$	A_0
A_2	$0 A_2$	0	$B_2 A_1$	0	$B_1 A_0$	0	0
0	0	$0 A_1$	0	$B_2 A_0$	0	$B_1 0$	B_1
A_1	$0 A_1$	0	$0 A_0$	0	$B_2 0$	0	0
0	0	$0 A_0$	0	0	0	$B_2 O$	
A_0	$0 A_0$	0	0	0	0	0	
	0	0	0	0	0	0	

Zadatak br. 4a

Grafički predstaviti tok podataka kod množenja 5–bitnih neoznačenih celih brojeva na protočnom Milerovom množaču. Kakve su performanse ovog množača ako se obavlja M uzastopnih množenja bez akumulacije, a kakve su performanse ako se množenje obavlja sa akomulacijom.

Cena leča je 3, sabirača 15 a AND kola 4.

Kašnjenje kroz leč je 2, kroz sabirač 10, a kroz AND kolo 5

Rešenje zadatka br. 4a

Prvo treba nacrtati onu čuvenu tabelu za prikaz toka podataka. Pošto se množe 5–bitni brojevi tabele treba da ima 10 kolona. Dakle,

	P9	P8	P7	P6	P5	P4	P3	P2	P1	P0	
B_0	$B_0$0	0	0	0	0	0	0	0	0	0	0
0	0	$B_0$0	0	0	0	0	0	0	0	0A_4	A_4
B_1	$B_1$0	0	$B_0$0	0	0	0	0	0	0A_4	0	0
0	0	$B_1$0	0	$B_0$0	0	0	0	0A_4	0	0A_3	A_3
B_2	$B_2$0	0	$B_1$0	0	$B_0$0	0	0A_4	0	0A_3	0	0
0	0	$B_2$0	0	$B_1$0	0	B_0A_4	0	0A_3	0	0A_2	A_2
B_3	$B_3$0	0	$B_2$0	0	B_1A_4	0	B_0A_3	0	0A_2	0	0
0											A_1
B_4											0
0											A_0
A_4											0
0											
A_3											
0											
A_2											
0											
A_1											
0											

A_0											

Tabela nije dovršena, ali pretpostavljam da je osnovni princip prikazan i da sami možete da nastavite.

A sada treba odrediti performanse ovog sistema ako se znaju latencije i cene za svaki element pojedinačno. Dakle, za slučaj bez akomulacije upotrebićemo formule:

a) bez akumulacije:

$T_C = T_L + T_{AND} + T_{ADD} = 17$

$C = (4N - 1) * T_C$

Za N = 5 (petostepeni protočni sistem) se dobija:

C = 19 * 17 = 323

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(3n)T_C}{M}\Big|_{n=5} = 255 + \frac{68}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 255$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=5} = 310 \cdot \tau(M) = 79050 + \frac{21080}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 79050$$

a) sa akumulacijom:

$T_C = T_L + T_{AND} + T_{ADD} = 17$

$C = (4N - 1) * T_C$

Za N = 5 (petostepeni protočni sistem) se dobija:

C = 19 * 17 = 323

Sada kada znamo cenu i kašnjenje možemo početi izraunavnje vremena potrebnog da se izvrši jedna instrukcija pri poslu od ukupno M instrukcija.

Dakle,

$$\tau(M) = \frac{(4n-1)T_C + (M-1)(2n-1)T_C}{M}\Big|_{n=5} = 153 + \frac{170}{M}$$

$$\tau = \lim_{M\to\infty} \tau(M) = 153$$

$$\mu(M) = (4C_L + C_S + C_{AND})2n \cdot \tau(M)\Big|_{n=3} = 47430 + \frac{52700}{M}$$

$$\mu = \lim_{M\to\infty} \mu(M) = 47430$$

Zadatak br. 5

Projektovati protočni sistem za izračunavanje funkcije f(x) kao rezultat pete iteracije u rekurentnoj formuli:

$$x_i = x_{i-1} \cdot \left(x_0 + \frac{x_0 + x_{i-1} \cdot x_0}{x_{i-1}^2} + x_{i-1} \right)$$

tako da se na izlazusvake ćelije, odnosno stepena sistema, dobija rezultat jedne iteracije. U okviru ćelije obezbediti vremenski optimalnu dvostepenu protočnost. Na raspolaganju je dovoljan broj sabirača, množača, delitelja i lečeva. Odrediti performanse ovako projektovanog protočnog sistema, ako se uzastopno izračunava M vrednosti date funkcije.

Napomena:

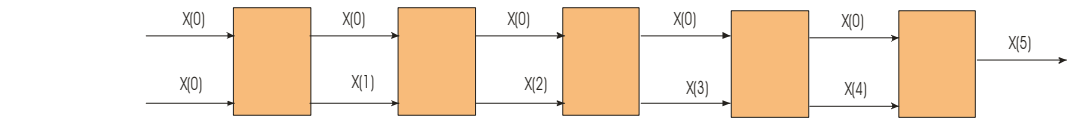
$C_{ADD}=16$ $T_{ADD}=5$

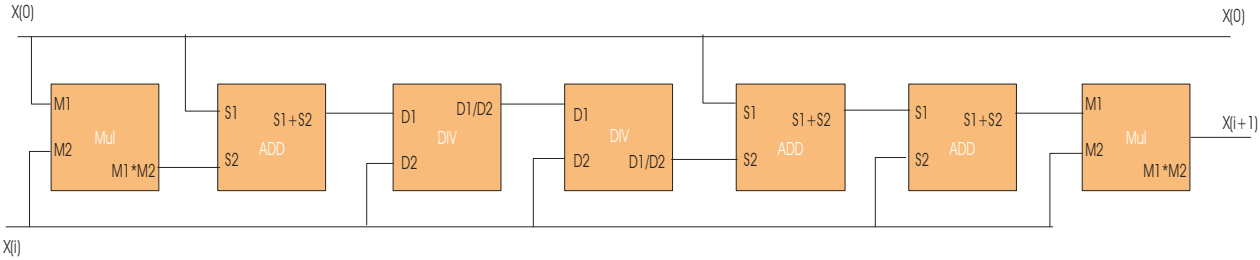
$C_{MUL}=20$ $T_{MUL}=10$

$C_{DIV}=24$ $T_{DIV}=15$

$C_L=3$ $T_L=2$

Rešenje zadatka br. 5





U slučaju ovakve ćelije, dvostepena protočnost se može obezbediti na sledeće načine:

MUL1 ADD,DIV1,DIV2,ADD2,ADD3,MUL2

MUL1,ADD1 DIV1,DIV2,ADD2,ADD3,MUL2

MUL1,ADD1,DIV1 DIV2,ADD2,ADD3,MUL2

MUL1,ADD1,DIV1,DIV2 ADD2,ADD3,MUL2

MUL1,ADD1,DIV1,DIV2,ADD2 ADD3,MUL2

MUL1,ADD1,DIV1,DIV2,ADD2,ADD3 MUL2

Od svih ovih mogućih slučajeva vremenski optimalna varijanta je ona koja ima najmanji taktni period:

$T_C=\max(T_L, T_{prvog\ stepena}, T_{drugog\ stepena})$

$T_{C1}=T_L+\max(T_{MULT},\ 3T_{ADD}+2T_{DIV9}+T_{MUL})=2+\max(10,55)=57$

$T_{C2}=T_L+\max(T_{MULT}+T_{ADD},\ 2T_{ADD}+2T_{DIV9}+T_{MUL})=2+\max(15,50)=52$

$T_{C3}=T_L+\max(T_{MULT}+T_{ADD}+T_{DIV},\ 2T_{ADD}+T_{DIV9}+T_{MUL})=2+\max(30,50)=37$

$T_{C4}=T_L+\max(T_{MULT}+2T_{ADD}+T_{DIV},\ 2T_{ADD}+T_{MUL})=2+\max(45,20)=47$

$T_{C5}=T_L+\max(T_{MULT}+2T_{ADD}+2T_{DIV},\ T_{ADD}+T_{MUL})=2+\max(50,15)=52$

$T_{C5}=T_L+\max(T_{MULT}+3T_{ADD}+2T_{DIV},\ T_{MUL})=2+\max(55,10)=57$

Vremenski optimalna dvostepena protočnost ćelije imaće taktni period $T_C=2T_{C3}=74$. Znači, lečeve ćemo umetnuti ispred prvog modula MUL1 i ispred četvrtog modula DIV2.

Performanse protočnog sistema, sa pet ovakvih ćelija pri izvršavanju M uzasopnih vrednosti funkcije f(x) su:

$$\tau(M)=\frac{N\cdot T_C+(M-1)\cdot T_C}{M}=\Big|_{N=5}=74\frac{M+4}{M}$$

$$\tau=\lim_{M\to\infty}\tau(M)=74$$

$$\eta(M)=N(2C_{DIV}+2C_{MUL}+3C_{ADD}+5C_L)\tau(M)=55870\frac{M+4}{M}$$

$$\mu=\lim_{M\to\infty}\eta(M)=55870$$

$$\delta=N\cdot T_C\Big|_{N=5}=5\cdot 74$$

N=5 zato što ovaj protočni sistem ima 5 ćelija tj. računa rezultat kao petu iteraciju neke iterativne formule.

Zadatak br. 6

Projektovati protočni sistem za izračunavanje funkcije $\sqrt{X}$ kao rezultat treće iteracije u rekurentnoj formuli:

$$X(i)=\frac{1}{2}\left(X(i-1)+\frac{X(0)}{X(i-1)}\right)$$

na raspolaganju je dovoljan broj delitelja, množača, sabirača i lečeva.

Napomena:

$C_{ADD}=18$, $T_{ADD}=4$

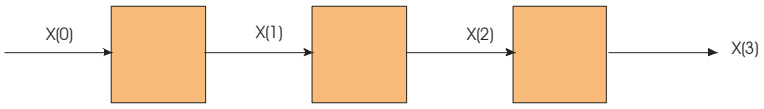
$C_{MUL}=20$, $T_{MUL}=8$

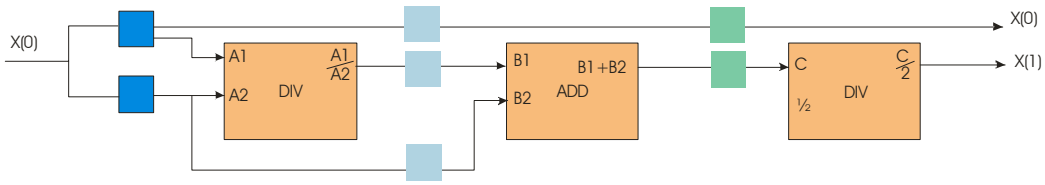
$C_{DIV}=20$, $T_{DIV}=10$

$C_L=3$, $T_L=2$

Rešenje zadatka br. 6

$$X(i)=\frac{1}{2}\left(X(i-1)+\frac{X(0)}{X(i-1)}\right)$$





Kada uvodimo protočnost, možemo se opredeliti za dvostepenu ili trostepenu protočnost. Dakle, u zavisnosti od našeg izbora dodaćemo svetlo plave lečeve za dvostepenu protočnost, a ukoliko se odlučimo za trostepenu dodaćemo i zelene lečeve (pored svtloplavih i tamno plavih).

Što se tiče optimalnosti za dvostepenu protočnost, možemo birati da li ćemo staviti svetlo zelene ili svetlo plave lečeve. Da bi izabrali optimalnije rešenje pribegavamo računici:

$T_{C1}=T_L+\max(T_D,T_A+T_D)=16$

$T_{C2}=T_L+\max(T_D+T_A,T_D)=16$

Iz ovoga zaključujemo da je potpuno sve jedno gde ćemo staviti lečeve. Dakle, za dvostepenu protočnost je potpuno sve jedno gde ćemo staviti lečeve jer se ništa ne menja što se kašnjenja tiče.

Ali, moramo razmotriti i kašnjenje u trostepenoj protočnosti.

Dakle,

$T_C=T_C+\max(T_D,T_A,T_D)=12$

Ovo je očigledno manje od kašnjenja za dvostepenu protočnost, dakle, ukoliko želimo optimalnije rešenje izabraćemo trostepenu protočnost.

No, pošto u zadatku ništa nije specificirano, uradićemo i za dvostepenu i za trostepenu protočnost.

Za dvostepenu protočnost:

$$\tau(M)=\frac{N\cdot 2\cdot T_C+(M-1)\cdot T_C}{M}=\Big|_{N=3}=16\frac{M+5}{M} \qquad \tau=\lim_{M\to\infty}\tau(M)=16$$

$$\eta(M)=N(5C_L+2C_D+C_A)\tau(M)=3504\frac{M+5}{M} \qquad \mu=\lim_{M\to\infty}\eta(M)=3504$$

Za trostepenu protočnost:

$$\tau(M)=\frac{N\cdot 3\cdot T_C+(M-1)\cdot T_C}{M}=\Big|_{N=3}=12\frac{M+8}{M}$$

$$\tau=\lim_{M\to\infty}\tau(M)=12$$

$$\eta(M)=N(7C_L+2C_D+C_A)\tau(M)=2844\frac{M+8}{M}$$

$$\mu=\lim_{M\to\infty}\eta(M)=2844$$

Ovim završavamo analizu. Ovaj zadatak je urađen potpuno poanalogiji sa prethodnim ;)

Zadatak br. 7

Projektovati protočni sistem za izračunavanje funkcije $f(x)=X_5$ po rekurentnoj formuli:

$$X(i)=\frac{X_{i-1}\cdot X_{i-2}+X_{i-1}}{X_{i-2}^{2}} \quad i=1,2,3,3,4\ ..$$

Početne vrednosti ovog niza su: $X_o=X_1=X$

Na izlazu svake ćelije (stepena) treba da se dobije rezultat jedne iteracije. U okviru ćelije obezbediti vremensku optimalnost trostepene protočnosti. Na raspolaganju je dovoljan broj sabirača, množača, delitelja i lečeva.

Odrediti perfotrmanse ovako projektovanog protočnog sistema ako se uzastopno izračunava M vrednosti funkcije f(x).

Cene i kašnjenja odgovarajućih elemenata su:

$C_{ADD}=16$, $T_{ADD}=4$

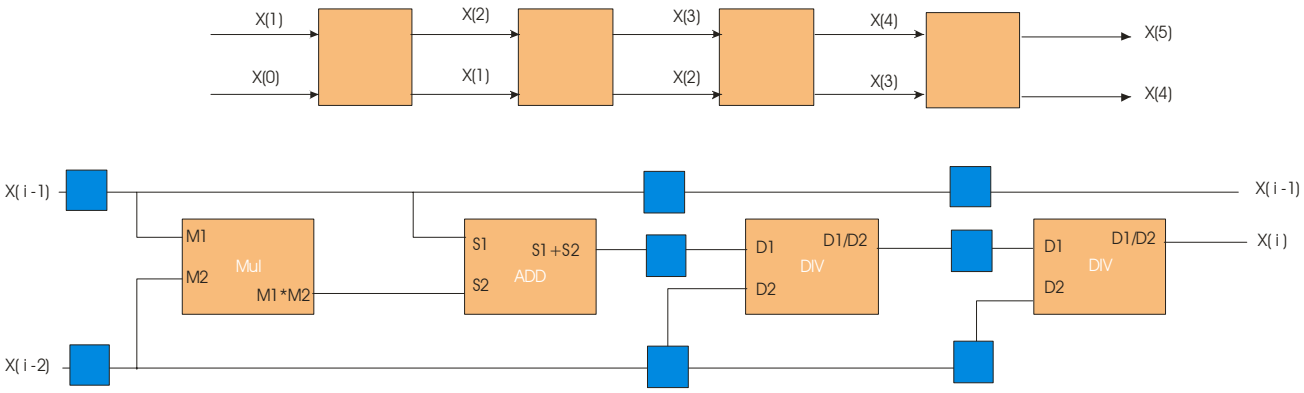
$C_{MUL}=20$, $T_{MUL}=8$

$C_{DIV}=20$, $T_{DIV}=10$

$C_L=3$, $T_L=2$

Pre nego što počnemo da projektujemo sistem, treba uočiti da trenutna vrednost zavisi od predhodne dve. Dakle, $x_2=f(x_1,x_0)$.

Rešenje zadatka br. 7



Kao i u prošlom zadatku, treba pronaći optimalno mesto gde treba staviti lečeve. Dakle, to znači da treba izabrati protočne stepene između kojih protočnih stepena treba postaviti lečeve da bi rešenje bilo optimalno. Dakle, pođimo redom:

Umetanje lečeva predstavićemo na sledeći način:

1. (MUL)	(ADD)	(DIV,DIV)
2. (MUL)	(ADD,DIV)	(DIV)
3. (MUL,ADD)	(DIV)	(DIV)

Sada treba analizirati:

1. $T_{C1}=T_L+max+(T_{MUL},T_{ADD},2T_{DIV})=22$
2. $T_{C2}=T_L+max(T_{MUL},T_{ADD}+T_{DIV},T_{DIV})=16$
3. $T_{C3}=T_L+max(T_{MUL}+T_{ADD},T_{DIV},T_{DIV})=14$

Očigledno je kašnjenje najmanje za situaciju opisanu pod rednim brojem 3.

$$\tau(M)=\frac{N\cdot 3\cdot T_C+(M-1)\cdot T_C}{M}=\Big|_{N=4}=\frac{12\cdot 14+(M-1)\cdot 14}{M}=14\frac{1+M}{M}$$

$$\tau=\lim_{M\to\infty}\tau(M)=14$$

$$\eta(M)=N(8C_L+C_{MUL}+C_{ADD}+2C_{DIV})\tau(M)=5600\frac{1+M}{M}$$

$$\mu=\lim_{M\to\infty}\eta(M)=5600$$

$\delta=3NT_C=168$ tri, zato što je sistem trostepen !!!!

Zadatak br. 8

Za dvostepeni protočni sistem prikazan na slici odrediti format mikroinstrukcije, zatim simbolički predstaviti rad svih stepena u sistemu za dve iteracije (I, I+1) programa:

FOR i=1 TO n DO

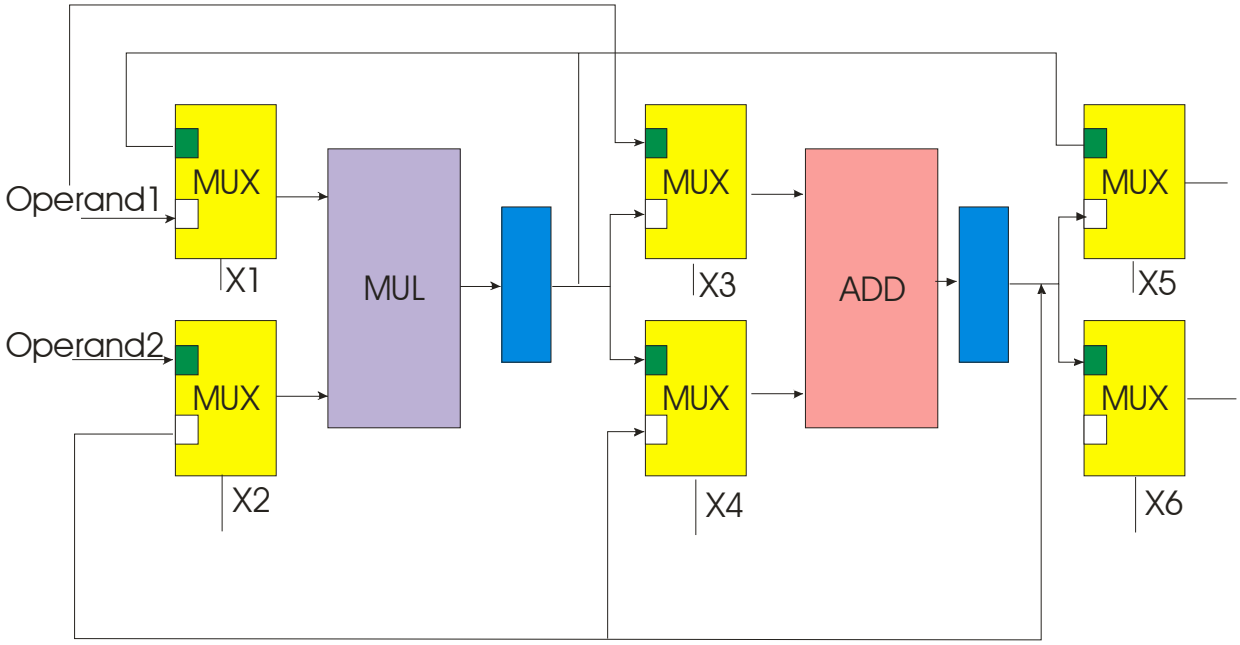
Y(i)=(A(i)*B(i)*C(i)+d(i)+e(i))+f(i)*g(i)

Ili pregledini napisano:

$$y_i = (a_i \cdot b_i \cdot c_i + d_i + e)_i + f_i \cdot g_i$$

I na kraju, napisati mikroprogram koji odgovara izvršenju malopre definisanom delu programa.

Rešenje zadatka br. 8



Zelenom bojom na multiplekserima prikazan je ulaz 0 a belom ulaz 1.

Dakle:

Operand1	Operand2	X1	X2	X3	X4	X5	X6

Operandi su brojevi nad kojm se u zavisnosti od kombinacije bitova X1....X6 izvršava neka operacija. Koja će to operacija biti, zavisi isključivo od kombinacije bitova X1...X6.

Sada, treba predstaviti šta se dešava u stepnima (ADD i MUL stepen).

Korak	MUL	ADD
1	$1 \cdot a_i$	$0+0$
2	$b_i \cdot a_i$	$d_i + 0$
3	$c_i \cdot a_i \cdot b_i$	$e_i + d_i$
4	$g_i \cdot f_i$	$c_i \cdot a_i \cdot b_i + e_i + d_i$
5	$0 \cdot 0$	$(c_i \cdot a_i \cdot b_i + e_i + d_i) + g_i \cdot f_i$
6	$1 \cdot a_{i+1}$	$0+0$
7	$b_{i+1} \cdot a_{i+1}$	$d_{i+1} + 0$
8	$c_{i+1} \cdot a_{i+1} \cdot b_{i+1}$	$e_{i+1} + d_{i+1}$
9	$g_{i+1} \cdot f_{i+1}$	$c_{i+1} \cdot a_{i+1} \cdot b_{i+1} + e_{i+1} + d_{i+1}$
10	$0 \cdot 0$	$(c_{i+1} \cdot a_{i+1} \cdot b_{i+1} + e_{i+1} + d_{i+1}) + g_{i+1} \cdot f_{i+1}$
11	$1 \cdot a_{i+2}$	$0+0$

Ufff, izgleda komplikovano zar ne?

Setimo se šta nama ustvari treba. Da, da to piše u postavci zadatka, ali je veoma nepregledno. Hajde da sada napišemo isto to, samo malo preglednije.

$$y_i = (a_i \cdot b_i \cdot c_i + d_i + e)_i + f_i \cdot g_i$$

Dakle, u **prvom koraku** treba da u sistem uvedemo A_i. To ćemo uraditi tako što ćemo selektovati ulaz sa prvog multipleksera sa koga dolazi operand1. Drugim rečima, podesićemo x1 na 1. Drugi operand je određen (multipleksiran) promenljivom x2, tako da ona treba ostati na 0, da bi bila selektovana linija koja nam dovodi operand2. Naravno, mi ćemo kao operand2 dovesto jedinicu, jer nam treba množenje Ai sa jedinicom.

Što se tiče drugog stepena u prvom koraku, tu je sve jasno. Sabiraju se dve nule. Te nule ćemo obezbediti tako što ćemo multipleksere koji dovode operande na sabirač podesiti tako da nam dovode dve nule. To ćemo izvesti tako što x3 podesimo na 1, a x4 podesimo na 0. Ovom kombinacijom bitova selektovaće se izlaz iz množača. Da li ste zbunjeni ? Da, selektuje se izlaz iz množača, jer je u tom trenutku tamo nema signala (logička nula), ili drugim rečima, kada nam treba nula, selektujemo ono za šta smo sigurni da nije pod naponon) nema signala).

U **drugom koraku** u sistem se uvodi B_i na operand2 liniju. B_i trebamo pomnožiti sa A_i (iz leča).

Na liniju operand1 se dovodi D_i. To D_i se treba dovesti na sabirač i sabrati sa nulom. Postavlja se pitanje kako se to izvodi ?

Vrlo prosto!

Da bi smo doveli B_i na operand 2 liniju dovoljno je da podesimo x2 liniju na 0. A_i se iz leča dovodi na ulaz množača tako što se stavi da je x1=0. Time smo obezbedili aktuelne operande za MUL stepen.

Što se tiče ADD stepena, na operand1 liniju treba dovesti D_i, a to se postiže tako što x3 podesimo na 0. Što se tiče drugog operanda za sabirački stepen, on treba ostati nula a to ćemo postići ako x4

podesimo na 1.

U **trećem koraku** u sistem treba uvesti C_i na operand2 i pomnožiti sa malopređašnjim proizvodom (A_i*B_i). To ćemo uraditi tako što ćemo podesiti x2 na 0, i x1 na 0.

Što se tiče ADD stepena, tamo treba uvesti E_i na operand1 liniju i sabrati ga sa D_i. To ćemo uraditi tako što ćemo podesiti x3 na 0 i x4 na 1 (time se selektuje predhodni izlaz iz sabirača.

Nadam se da je princip objašnjen i sa ova tri koraka. Sada ćemo ispisati konažno rešenje zadatka, što se mikroprograma tiče.

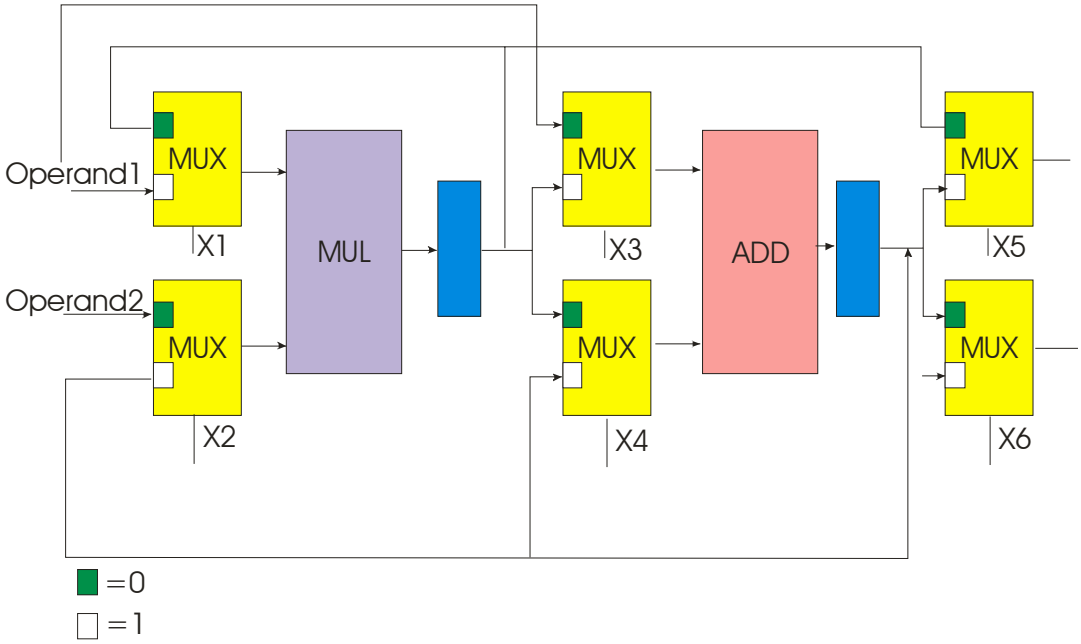
	Operand1	Operand2	X1	X2	X3	X4	X5	X6
1	A_i	1	1	0	1	0	X	X
2	D_i	B_1	0	0	0	1	X	X
3	E_I	C_I	0	0	0	1	X	X
4	G_I	F_I	1	0	1	1	X	X
5	0	0	1	0	1	1	X	X
6	A_{i+1}	1	1	0	1	0	1	0
7	D_{i+1}	B_{1+1}	0	0	0	1	X	X
8	E_{I+1}	C_{I+1}	0	0	0	1	X	X
9	G_{I+1}	F_{I+1}	1	0	1	1	X	X
10	0	0	1	0	1	1	X	X
11	A_{i+2}	1	1	0	1	1	1	0

Zadatak br. 9

Za dvostepeni protočni sistem prikazan na slici odrediti format mikronaredbe. Simboližki predstaviti rad svih stepena u sistemu i napisati mikroprogram za slučaj izvršavavanja dveju iteracija sledećeg programa.

FOR i=1 to N DO

$A_i=(b_i^2*c_i+d_i+3)+e_i*f_i$



Rešenje zadatka br. 9

Zelenom bojom na multiplekserima prikazan je ulaz 0 a belom ulaz 1.

Prvo treba napisati format mikroinstrukcije. Služićemo se istom logikom kao u prošlom zadatku. Dakle:

Operand1	Operand2	X1	X2	X3	X4	X5	X6

Operandi su brojevi nad kojm se u zavisnosti od kombinacije bitova X1....X6 izvršava neka operacija. Koja će to operacija biti, zavisi isključivo od kombinacije bitova X1...X6.

Sada, treba predstaviti šta se dešava u stepnima (ADD i MUL stepen).

Korak	MUL	ADD
1	$1 \cdot b_i$	$0+0$
2	$b_i \cdot b_i$	$d_i + 0$
3	$c_i \cdot b_i^2$	$3 + d_i$
4	$c_i \cdot f_i$	$\left(c_i \cdot b_i^2\right) + \left(d_i + 3\right)$
5	$0 \cdot 0$	$\left(c_i \cdot b_i^2\right) + \left(d_i + 3\right) + e_i f_i$
6	$1 \cdot b_{i+1}$	$0+0$
7	$b_{i+1} \cdot b_{i+1}$	$d_{i+1} + 0$
8	$c_{i+1} \cdot b_{i+1}^2$	$3 + d_{i+1}$
9	$c_{i+1} \cdot f_{i+1}$	$\left(c_{i+1} \cdot b_{i+1}^2\right) + \left(d_{i+1} + 3\right)$
10	$0 \cdot 0$	$\left(c_{i+1} \cdot b_{i+1}^2\right) + \left(d_{i+1} + 3\right) + e_i$
11	$1 \cdot b_{i+2}$	$0+0$

	Operand1	Operand2	X1	X2	X3	X4	X5	X6
1	B_i	1	1	0	1	0	X	X
2	D_i	B_i	0	0	0	1	X	X
3	3	C_i	0	0	0	1	X	X
4	C_i	F_i	1	0	1	1	X	X
5	0	0	1	0	1	1	X	X
6	B_{i+1}	1	1	0	1	0	1	0
7	D_{i+1}	B_{i+1}	0	0	0	1	X	X
8	3	C_{i+1}	0	0	0	1	X	X
9	C_{i+1}	F_{i+1}	1	0	1	1	X	X
10	0	0	1	0	1	1	X	X
11	B_{i+2}	1	1	0	1	1	1	0

ZADATAK BR. 10

Projektovati dvostepeni protočni sistem za izračunavanje elemenata niza:

$$x_i = x_{i-1}(a_i + 1) + a_i(b_i + 1) \quad X_0=0$$

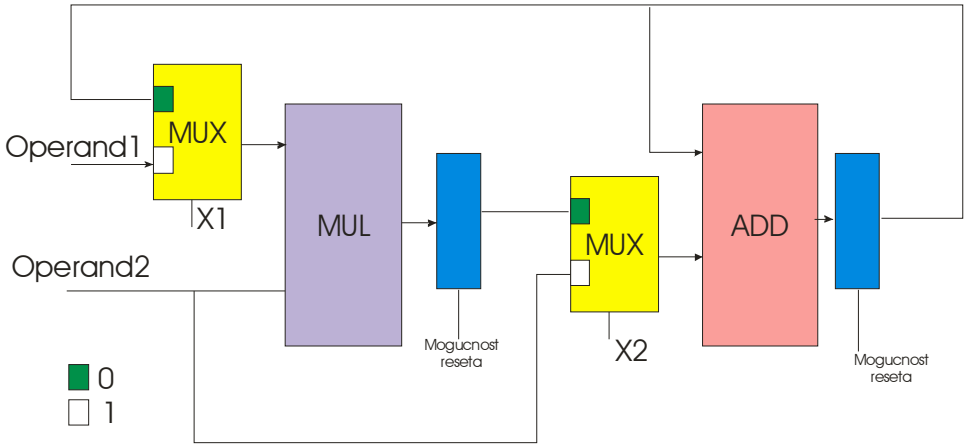
Na raspolaganju su dva procesna elementa bez lokalnih memorija, dva multuipleksera 2x1 i dovoljan broj lečeva, uz mogućnost reseta sistema. Jedan procesni element uvek obavlja operaciju sabiranja, a drugi operaciju množenja. Ko nenulti operandi na ulazu mogu se javiti samo elementi nizova $A(a_0 \ldots a_n)$ i $B(b_0 \ldots b_n)$.

Napisati format mikronaredbe za ovakav sistem i napisati mikroprogram za izvršavanje dela programa kojim se izračunava prvih 4 elemenata niza X.

Rešenje zadatka br. 10

Moramo malo srediti početni izraz

$$x_i = x_{i-1}(a_i + 1) + a_i(b_i + 1) = x_{i-1} \cdot a_i + x_{i-1} + a_i \cdot b_i + a_i$$



Pokušaćemo da napišemo odgovarajući format mikro naredbe.

Operand1	Operand2	x1	X2

Op1	Op2	x1	x2	Rezultat1	Rezultat2
0	0	x	1	0*0	$0+0=X_0$
0	A_1	0	1	X_0*A_1	X_0+A_1
B_1	A_1	1	0	B_1*A_1	$X_0A_1+X_0+A_1$
0	0	X	0	0*0	$B_1A_1+X_0A_1+X_0+A_1=X_1$
0	A_2	0	1	X_1A_2	X_1+A_2
B_2	A_2	1	0	B_2A_2	$X_1A_2+X_1+A_2$
0	0	X	0	0*0	$B_2A_2+X_1A_2+X_1+A_2=X_2$
0	A_3	0	1	X_2A_3	X_2+A_3
B_3	A_3	1	0	B_3A_3	$X_2A_3+X_2+A_3$
0	0	X	0	0*0	$B_3A_3+X_2A_3+X_2+A_3=X_3$

Zadatak br. 11

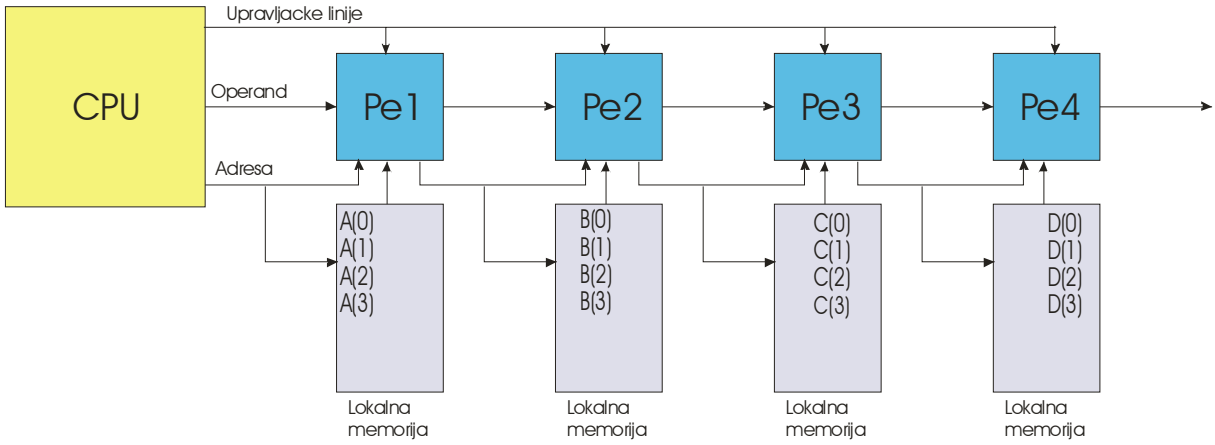
Na slici je data programirljiva protočna ALU sa fiksnom strukturom. Svaki procesni element može da obavlja operacije množenja, deljenja, sabiranja i oduzimanja. Naravno, može i da obavlja operaciju NOP. Operandi i rezultati su 32 bitni a lokalne memorije su kapaciteta 1K 32 bitnih reči.

Odrediti strukturu mikro operacije za ovakvu ALU

Napisati program za izvršenje dela programa:

FOR i=0 TO 3 DO

E(i)=((a(i)*a(i)*b(i))/c(i))*d(i)



Rešenje zadatka br. 11

Da bi lakše vidli o kom se izrazu radi, napisaćemo ga u vidu razlomaka.

$$e(i) = \frac{a(i) \cdot a(i) \cdot b(i) \cdot d(i)}{c(i)}$$

Sada je mnogo lakše da vidimo koje se to ustvari operacije obavljaju zar ne?

Hajde sada da malo razjasnimo kako funkcioniše ova protočna ALU.

Sa upravljačke linije se dovodi operacija koja se izvršava nad operandom. Operand se čita preko linje koja vodi iz lokalne memorije.

Šta je sa drugim operandom?

Pa naravno, samo jedan operand je iz lokalne memorije, a drugi je iz rezultat iz predhodnog stepena. Adresa lokalne memorije je prenesena sada u drugi stepen, a u prvi je unešena nova (iz mikroinstrukcije). Dakle, primećujemo da podaci teku kroz protočni sistem.

Hajde sada da odredimo format mikroinstrukcije:

Operand	Adresa	Op1	Op2	Op3	Op4

Pošto je u zadatku naglašeno da su operandi 32 bitni brojve, onda polje Operand zauzima 32b. Što se tiče polja adresa, u zadatku se kaže da ima 1K adresa, a to znači da je doljno 10 bitova da se adresira

1024 adresa)2^{10}=1024. Dakle, polje adresa mora da ima 10 bitova. Što se tiče operacija, pošto ih ima 5, moramo izdvojiti minimalno 3 bita da bi jednoznačno označili operacije. Dakle, polje op ima 3 bita. Pošto ima 4 protočna stepena, onda i ovih polja mora biti 4.

Dakle, kao zaključak možemo napisati:

32b	10b	3b	3b	3b	3b

$$e(i)=\frac{a(i)\cdot a(i)\cdot b(i)\cdot d(i)}{c(i)}$$

Sada, kada smo našli format mikronaredbe, možemo početi da pišemo mikro program.

	Op.	Adre.	Op1	Op2	Op3	Op4	REZ1	REZ2	REZ3	REZ4	LM1	LM2	LM3	LM4
1	A(0)	ADR	MUL	NOP	Nop	Nop	A_0A_0	X	X	X	A_0	B_0	C_0	D_0
2	A(1)	ADR+1	MUL	MUL	Nop	Nop	A_1A_1	$A_0A_0B_0$	X	X	A_1	B_1	C_1	D_1
3	A(2)	ADR+2	MUL	MUL	DIV	Nop	A_2A_2	$A_1A_1B_1$	$A_0A_0B_0/C_0$	X	A_2	B_2	C_2	D_2
4	A(3)	ADR+3	MUL	MUL	DIV	MUL	A_3A_3	$A_2A_2B_2$	$A_1A_1B_1/C_1$	$A_0A_0B_0D_0/C_0$	A_3	B_3	C_3	D_3
5	X	X	Nop	MUL	DIV	MUL		$A_3A_3B_3$	$A_2A_2B_2/C_2$	$A_1A_1B_1D_1/C_1$				
6	X	X	Nop	Nop	DIV	MUL			$A_3A_3B_3/C_3$	$A_2A_2B_2D_2/C_2$				
7	X	X	Nop	Nop	Nop	MUL				$A_3A_3B_3D_3/C_3$				

Hajmo sada da posmatrajući ovo rešenje pokušamo da ga prokomentarišemo!

Krenimo redom.

Prva mikro naredba:

Ona nam šalje A(0) kao operand,

Što se tiče adrese, ona nam šalje neku adresu od koje počinju podaci u lokalnoj memoriji (lokalnim memorijama). Pošto nama treba A(0) na kvadrat, u lokalnu memoriju procesnog elemnta PE1 treba staviti A(0) na adresu ADR.

Zatim, ona nam govori da treba POMNOŽITI operand A(0) sa operandom koji se nalazi na adresi ADR u lokalnoj memoriji prvog procesnog elementa. Uostalom, ovo množenje se svakako odnosi na prvi procesni element, jer je naredba za množenje izdata u polju op1. Sada možemo ponovo naglasiti da se op1, op2, op3, op4 odnose na upravo te procesne elemente. Dakle, op1 na PE1, op2 na PE2, op3 na PE3 i op4 na PE4.

Dok još uvek komentarišemo prvu mikronaredbu, moramo napomenuti da ona ostavlja procesne elemente PE2, PE3 i PE4 nezapošljene, jer im je prosledila operaciju NOP, koja ustvari znači No Operation (nema operazije za tebe !).

Rezultat prve naredbe:

Rezultat prve mikronaredbe će biti množenje operanda A(0) sa operandom koji se nalazi na adresi ADR u lokalnojmemoriji prvog procesnog elementa. Svi ostali procesni elementi otaće nezaposleni.

E, sada mi možemo da biramo šta će da se nalazi u lokalnim memorijama.

Ako pogledamo izraz iz postavke zadataka, primetićemo da je potrebno pomnožiti A(i) sa A(i) znači naći kvadrat. Dakle, ako je kao operand prosleđen A(0), u memoriji se treba naći takoće A(0). Nadam se da je potpuno jasno zašto je to tako.

Druga naredba:

Druga naredba nam prosleđuje operand A(1) procesnom elementu PE1. Što se adrese tiče, i ona se inkrementira i pokazuje sada na adresu ADR+1. Primećujemo da se u drugoj naredbi upotrebljavaju i prvi i drugi procesni element. Ovo smo zaključili na osnovu dve MUL operacije koje su izdate i za PE1 i PE2.

šta se dalje dešava?

Prvi procesni element množi sadržaj svoje memorije sa lokacije ADR+1 sa operandom koji mu je prosleđen, a to je A(1).

Drugi procesni element vrši množenje, ali, jedan operand mu je rezultat prethodnog koraka procesnog elementa PE1, a drugi mu se nalazi na adresi ADR. Znači, on prihvata adresu kojom se bavio prethodni protočni stepen u prošlom trenutku, a kao operand uzima rezultat predhodnog protočnog stepena u prošlom trenutku.

Rezultat druge naredbe:

Dakle, prvi procesni element će već početi da množi A(1) sa sadržajem lokalne memorije na adresi ADR+1. Ukoliko znamo da nam ponovo treba kvadrat A(i) operanda, onda ćemo u lokalnu memoriju na adresu ADR+1 staviti takođe A(1), čijim ćemo množenjem sa A(1) dobiti kvadrat.

Drugi procesni element će pomnožiti $A(0)^2$ sa sadržajem lokalne memorije na adresi ADR. Dakle, na tu dresu treba da stavimo B(0) jer nam to treba (pogledaj postavku zadatka).

Nadam se da su komentari prve dve naredbe uticali na bolje razumevanje načina na kome funkcioniše ovaj protočni sistem i njegov mikro program. Dalju analizu možete samostalno nastaviti zar ne? :)

Zadatak br. 12

Projektovati dvostepeni protočni sistem za izračunavanje elemenata niza $x_i=a_i*(x_{i-1}+d_i)+b_i$. na raspolaganju su dva procesna elementa koja mogu da obavljaju operacije MUL i ADD. Procesni elementi nemaju lokalnu memoriju. Sistem ne poseduje mogućnost reseta, i osim raspoloživih procesnih elemenata nije dozvoljeno korišćenje nikakvg dodatnog hardvera.

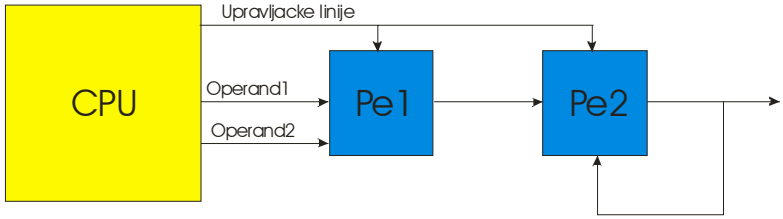
Napisati format instrukcije za dati sistem i napisati mikroprogram za izvršavanje dela programa:

FOR i=2 to 4 DO

$X_i=a_i*(x_{i-1}+d_i)+b_i$,$x_1=1$

Rešenje zadatka br. 12

Pošto je nezamislivo da nema lečeva, onda ćemo naglasiti da se lečevi nalaze u procesnim elementima.



vrlo je važno primetiti da postoje veze povratne sprege. Naravno, povratna sprega će se uvek javljati kada imamo zavisnost tekućeg elementa od nekog predhodnog.

Naravno, nemožemo a da ne pomenemo da pošto nema mogućnosti reseta, reset ćemo izvest tako što ćemo propustiti sve nule kroz protočni sistem.

Format mikroinstrukcije je:

Operand1	Operand2	Ko1	Ko2

Hajdemo sada da opišemo šta se dešava u protočnom sistemu. Naravno, kao i prošli put, treba prvo lepo pregledno napisati šta nama treba:

ustvari

$x_i = a_i \cdot (x_{i-1} + d_i) + b_i$; početna vrednost ovog niza je $X_1=1$

	PE1	PE2
1	$0 \cdot 0$	X
2	$a_2 \cdot x_1$	X
3	$a_2 \cdot d_2$	$a_2 \cdot x_1 + 0$
4	$b_2 \cdot 1$	$a_2 \cdot d_2 + a_2 \cdot x_1$
5	$a_3 \cdot 1$	$b_2 + a_2 \cdot d_2 + a_2 \cdot x_1 = x_2$
6	$a_3 \cdot d_3$	$a_3 \cdot x_2$
7	$b_3 \cdot 1$	$a_3 \cdot d_3 + a_3 \cdot x_2$
8	$a_4 \cdot 1$	$b_3 + a_3 \cdot d_3 + a_3 \cdot x_2 = x_3$
9	$a_4 \cdot d_4$	$a_4 \cdot x_3$
10	$b_4 \cdot 1$	$a_4 \cdot d_4 + a_4 \cdot x_3$
11	X	$b_4 + a_4 \cdot d_4 + a_4 \cdot x_3 = x_4$

Na osnovu ove tabele, lako možemo napisati mikro program.

	Op1	Op2	Ko1	Ko2
1	0	0	MUL	–
2	A2	1	MUL	MUL
3	A2	D2	MUL	ADD
4	B2	1	MUL	ADD
5	A3	1	MUL	ADD
6	A3	D3	MUL	MUL
7	B3	1	MUL	ADD
8	A4	1	MUL	ADD
9	A4	D4	MUL	MUL
10	B4	1	MUL	ADD
11	–	–	MUL	ADD

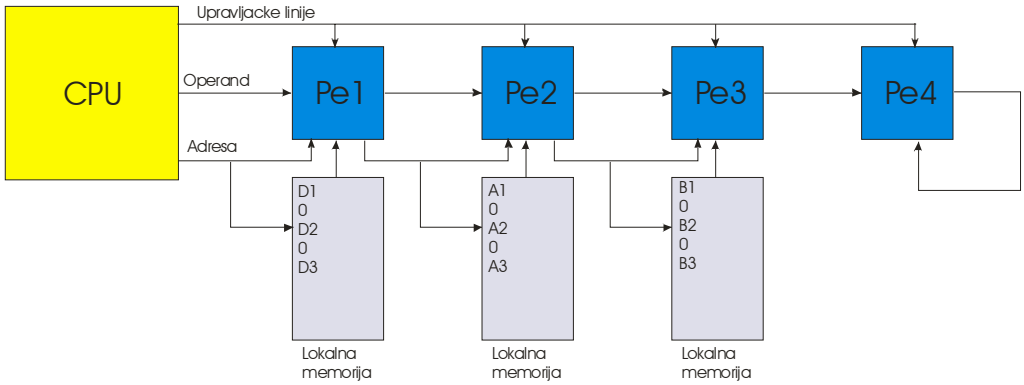
Zadatak br. 13

Projektovati četvorostepeni protočni sistem za izračunavanje izraza: $c_i = a_i(b_i + c_{i-1}) + d_i^2$; $c_0=0$

pri čemu svaki procesni element može da obavlja operacije +,–,*,/ i NOP. Na raspolaganju su procesni elementi sa i bez lokalne memorije.

Napisati format mikronaredbe za dati sistem a zatim napisati mikroprogram za izračunavanje prva tri elementa niza c.

Rešenje zadatka br. 13



	PE1	PE2	PE3	PE4
1	d_1^2	X	X	X
2	a_1	$\frac{d_1^2}{a_1}$	X	X
3	d_2^2	a_1	$\frac{d_1^2}{a_1} + b_1$	X
4	a_2	$\frac{d_2^2}{a_2}$	a_1	$\frac{d_1^2}{a_1} + b_1 + c_0$
5	d_3^2	a_2	$\frac{d_2^2}{a_2} + b_2$	$a_1 \cdot \left(\frac{d_1^2}{a_1} + b_1 + c_0 \right)$
6	a_3	$\frac{d_3^2}{a_3}$	a_2	$\frac{d_2^2}{a_2} + b_2 + c_1$

7		a_3	$\frac{d_3^2}{a_3}+b_3$	$a_2\cdot\left(\frac{d_2^2}{a_2}+b_2+c_1\right)$
8			a_3	$\frac{d_3^2}{a_3}+b_3+c_2$
9				$a_3\cdot\left(\frac{d_3^2}{a_3}+b_3+c_2\right)$

Kao i obično, i sada ćemo prvo napisati format mikroinstrukcije:

operand	adresa	Ko1	Ko2	Ko3	Ko4

Zatim, treba napisati šta treba da se dešava u procesnim elementima da bi se izvršilo trazeno izračunavanje. To će biti prikazano tabelom sa leve strane.

Na osnovu ove tabele lako se može napisati mikroprogram koji izgleda ovako:

	Op	Adresa	Ko1	Ko2	Ko3	Ko4	Rez1	Rez2	Rez3	Rez4	LM1	LM2	LM3
1	D_1	ADR	MUL	NOP	NOP	NOP	D_1^2	X	X	X	D_1	A_1	B_1
2	A_1	ADR+1	ADD	DIV	NOP	NOP	A_1+0	D_1^2/A_1	X	X	0		
3	D_2	ADR+2	MUL	ADD	ADD	NOP	D_2^2	A_1+0+0	D_1^2/A_1+B_1	X			
4	A_2	ADR+3	ADD	DIV	ADD	ADD	A_2+0	D_2^2/A_2	$A_1+0+0+0$	$D_1^2/A_1+B_1+C_0$			
5	D_3	ADR+4	MUL	ADD	ADD	MUL	D_3^2	A_2+0+0	D_2^2/A_2+B_2	$A_1(D_1^2/A_1+B_1+C_0)$			
6	A_3	ADR+5	ADD	DIV	ADD	ADD			$A_2+0+0+0$	$D_2^2/A_2+B_2+C_1$			
7	X	X	NOP	ADD	ADD	MUL				$A_2(D_2^2/A_2+B_2+C_1)$			
8	X	X	NOP	NOP	ADD	ADD							
9	X	X	NOP	NOP	NOP	MUL							

Primer nije potpuno završen. Dakle, ideja je bila da se pokaže osnovni princip funkcionisanja protočnog sistema, a na vama je da sami privedete kraju takozvani protok podataka.

Zadatak br. 14

Projektovati četvorostepeni protočni sistem za izračunavanje elemenata niza A, na osnovu izraza:

$$a_i = \frac{(c_i + e_i)\cdot(e_i - c_i)}{b_i \cdot c_i + d_i \cdot f_i^2}$$

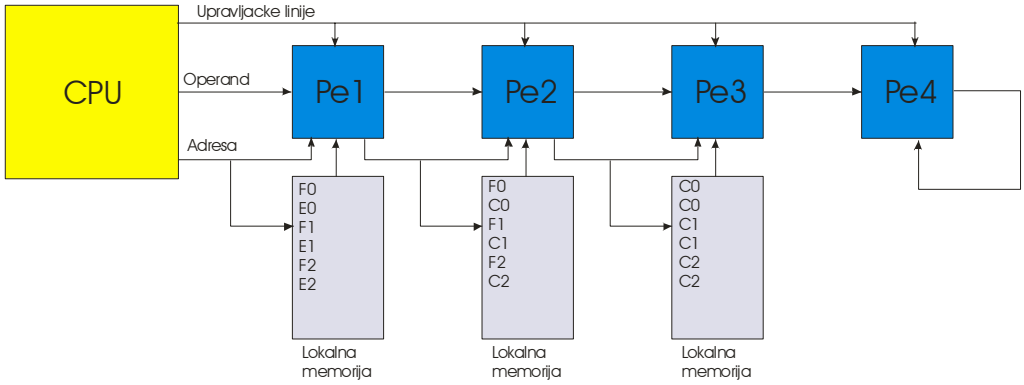
Svaki procesni element u sistemu može da obavlja osnovne aritmetičke operacije i naredbu NOP. Uvodi se pretpostavka da kada se izvršava naredba NOP, da se operand sa ulaza procesnog elementa prosleđuje na njegovizlaz. Pri projektovanju sistema ne koristiti kultipleksere.

Odrediti format mikronaredbe i simbolički predstaviti šta se dešava u procesnim elementima kada se izvršava deo programa koji računa članove niza A. Naravno, prikazati i sadržaj lokalnih memorija.

REŠENJE ZADATKA BR. 14

Možda bi bilo klakše kada bi transformisali početni iszraz u:

$$a_i = \frac{e_i^2 - c_i^2}{b_i \cdot c_i + d_i \cdot f_i^2} = \frac{\frac{e_i^2}{c} - c_i}{b_i + \frac{d_i \cdot f_i^2}{c_i}}$$



Odmah ćemo pisati mikroprogram:

OP	ADR.	Ko1	Ko2	Ko3	Ko4	REZ1	REZ2	REZ3	REZ4	LM1	LM2	LM3
B0	X	NOP	NOP	NOP	NOP	B_0	X	X	X	F_0	F_0	
D0	ADR	MUL	NOP	NOP	NOP	D_0F_0	B_0	X	X	E_0	C_0	C_0
E0	ADR+1	MUL	MUL	NOP	NOP	E_0E_0	$D_0F_0F_0$	B_0	X			
B1	X	NOP	DIV	DIV	NOP	B_1	E_0E_0/C_0	$D_0F_0F_0/C_0$	B_0			
B1	ADR+2	MUL	NOP	SUB	ADD	B_1F_1	B_1	E_0E_0/C_0-C_1				
E1	ADR+3	MUL	MUL	NOP	DIV							
B2	X	NOP	DIV	DIV	NOP							
D2	ADR+4	MUL	NOP	SUB	ADD							
E2	ADR+5	MUL	MUL	NOP	DIV							
X	X	NOP	DIV	DIV	NOP							
X	X	NOP	NOP	SUB	ADD							
X	X	NOP	NOP	NOP	DIV							

Zadatak br. 15

Projektovati trostepeni protočni sistem za izračunavanje elemenata niza D na osnovu rekurentne formule:

$$d_i = \frac{\frac{e_i + f_i}{c_i} \cdot h_i}{a_i^2 \cdot d_{i-1} - g_i \cdot b_i} \qquad d_0=0$$

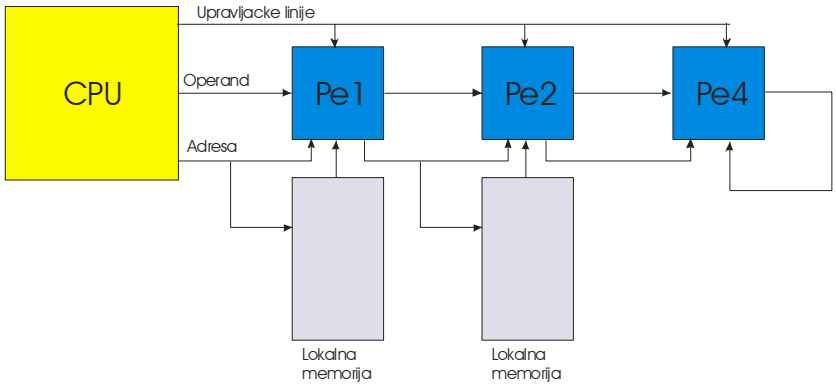
Svaki procesni element u sistemu može da obavlja atitmetičke operacije. Na raspolaganju su procesni elementi sa i bez lokalne memorije.

Odrediti format mikronaredbe opisanog sistema i simbolički prikazati izvršenje dela mikroprograma.

Narvno, prikazati i sadržaj lokalnih memorija pri izračunavanju prva tri elementa niza.

Rešnje zaradtka br. 15

Prvo treba nacrtati protočni sistem.



Format instrukcije ovakvog protočnog sistema će biti:

operand	adresa	Ko1	Ko2	Ko3

Pošto uovom slučaju ne može da se javinaredba NOP, već samo neke od naredbi (+,–,/,*) onda nam je dovoljno 2 bita da identifikujemo naredbu. Dakle polja KO mogu biti 2–bitna.

Kao i obižno, hajde da malo uprotimo izraz za koji smo projektovali protočni sistem. Obično preuređujemo izraz tako da imamo po nekoliko operacija koje će moći da se obavljaju simultano.

Dakle,

$$d_i = \frac{\frac{e_i + f_i}{c_i} \cdot h_i}{a_i^2 \cdot d_{i-1} - g_i \cdot b_i} = \frac{-\frac{e_i + f_i}{c_i}}{\frac{g_i \cdot b_i}{h_i} - \frac{a_i^2 \cdot d_{i-1}}{h_i}}$$

Pokušajmo sada na osnovu ovoga da napišemo mikro program, i u isto vreme da odredimo sadržaj lokalnih memorija.

Operand	Adresa	Ko1	Ko2	Ko3	REZULTAT1	REZULTAT2	REZULTAT3	LM1	LM2
A0	ADR	MUL	X	X	A0*A0	X	X	A0	H0
G0	ADR+1	MUL	DIV	X	G0*B0	A0*A0/H0	X	B0	H0
E0	ADR+2	ADD	DIV	MUL	E0+F0	G0*B0/HO	–C0*A0*A0/H0	F0	–C0
A1	ADR+3	MUL	DIV	SUB				A1	H1
G1	ADR+4	MUL	DIV	DIV				B1	H1
E1	ADR+5	ADD	DIV	MUL				F1	–C1
A2	ADR+6	MUL	DIV	SUB				A2	H2
G2	ADR+7	MUL	DIV	DIV				B2	H2
E2	ADR+8	ADD	DIV	MUL				F2	–C2

Zadatak br. 16

Projektovati trostepeni protočni sistem za izračunavanje elemenata niza C na osnovu formule:

$$c_i = a_i \cdot b_i + c_{i-1} \cdot d_i \qquad c_0=0$$

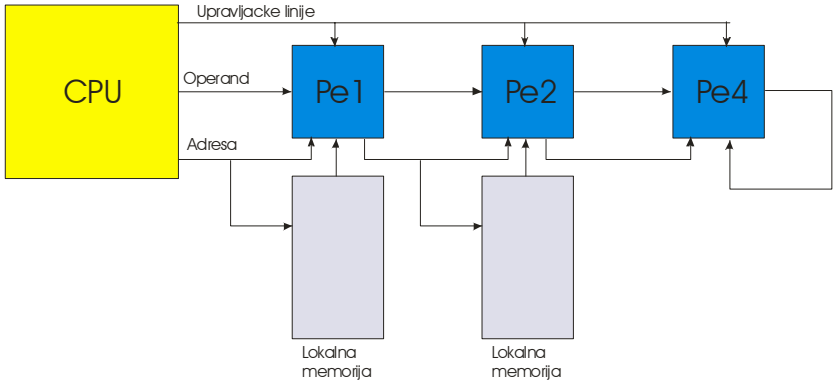
Svaki procesni element u sistemu može da obavlja osnovne atitmetičke operacije i naredbu NOP. Na raspolaganju su procesni elementi sa i bez lokalnih memorija.

Odrediti format mikronaredbe tog protočnog sistema i simbolički prikazati izvršenje dela mikroprograma, kao i sadržaj lokalnih memorija koji mu odgovara za izračunavanje prva tri elementa niza.

Rešenje zadatka br. 16

Napišimo sada polaznu formulu u malo drugačijem obliku. Zašto se uvek piše u nekom drugom obliku? Zato što, kada imamo neku rekurentnu vezu tj. u izrazu nam se pominje neko x_{i-1}, ukoliko imamo trostepeni protočni sistem, izraz moramo transformisati tako da ima 4 operacije, da bi ovo bilo moguće. Najbolje ćete to shvatiti na primeru!

$$c_i = a_i \cdot b_i + c_{i-1} \cdot d_i = \left(\frac{a_i \cdot b_i}{d_i} + c_{i-1} \right) \cdot d_i$$



Operand	Adresa	Ko1	Ko2	Ko3	REZULTAT1	REZULTAT2	REZULTAT3	LM1	LM2
A_1	ADR	MUL	NOP	NOP	A_1*B_1	X	X	B_1	D_1
D_1	ADR+1	ADD	DIV	NOP	D_1	A_1*B_1/D_1	X	0	0
A_2	ADR+2	MUL	ADD	ADD	A_2*B_2	D_1	A_1*B_1/D_1+C_0	B_2	D_2
D_2	ADR+3	ADD	DIV	MUL	D_2	A_2*B_2/D_2	$(A_1*B_1/D_1+C_0)D_1$	0	0
A_3	ADR+4	MUL	ADD	ADD	A_3*B_3	D_2	A_2*B_2/D_2+C_1	B_3	D_3
D_3	ADR+5	ADD	DIV	MUL	D_3	A_3*B_3/D_3	$(A_2*B_2/D_2+C_1)D_2$		
X	X	NOP	ADD	ADD	X	D_3	A_3*B_3/D_3+C_2		

X	X	NOP	NOP	MUL	X	X	$(A_3*B_3/D_3+C_2)D_3$		

Ovo je tabele koja sadrži i mikroprogram i međurezultate i sadržaj lokalnih memorija. Komentarišući ovu tabelu, pokušaću da objasnim način pisanja ovog programa.

Prva mikronaredba:

- U početku uopšte neznamo šta treba da radimo! Hajde sada ponovo da pogledamo izraz koji se traži! logično bi bilo da pođemo od onog A*B zar ne?
- Dakle, u prvoj mikronaredbi šaljemo operand A_1, zatim šaljemo adresu na kojoj se nalazi B_1 (naravno u lokalnoj memoriji prvog procesnog elementa). Možda ovo zbunjuje, trebalo je preformulisati rečenicu koja govori da se u lokalnoj memoriji nalazi B_1 u rečenicu TREBA da se nalazi (dakle mi biramo sadržaj memorije tj. u trenutku kada shvatimo da nam trebe neki podatak, mi ga jednostavno tada upišemo u tabelu sadržaja memorija. To ćemo i sada učiniti tj. upisati B_1 u lokalnu memoriju procesnog elementa PE1).
- Šaljemo operaciju MUL koja treba da se obavi u procesnom elementu PE1.
- Rezultat ovakve mikronaredebe će biti:
- Prvi procesni element će kao operande imati ulazni operand A_1 i operand sa memorijske lokacije ADR iz svoje lokalne memorije. Dakle, A_i i B_1. Ono što PE1 treba da uradi, je da pomnoži ova dva broja.
- Ostali procesni elementi neće raditi ništa tj. pošto ima dolazi NOP naredba, oni će u ovoj naredbi biti neiskorišćeni (To je označeno oznakama X u tabeli).

Druga mikronaredba:

- Sada se u procesni element PE1 šalje podatak D_1, u nameri da se dobije prvi sabirak u zagradi iz izraza koji nam treba.
- Zatim, procesnom elementu PE1 se šalje adresa ADR+1 na kojoj se nalazi 0 (nula) da bi bri izvršavanju naredbe ADD, rezultat prvog procesnog elementa bio ponovo D_1 (sabira se sa nulom da se ne bi promenio). Zašto se ovo radi razjasnićemo malo kasnije.
- Drugi procesni element PE2 treba da izvrši naredbu DIV. On tu naredbi izvršava nad rezultatom iz procesnog elementa PE1 iz prošlog koraka, dakle nad A_1*B_1 i nad sadržajem svoje lokalne memorije na adresi koja mu takođe dolazi iz predhodnog procesnog elementa u prošlom koraku. Dakle, D_1. Nesumnjivo je da će tezultat drugog procesnog elementa biti: A_1*B_1/D_1. Što se tiče trećeg procesnog elementa on je i dalje nezaposlen.

Treća mikronaredba:

- Kao operanda dovodimo A_2 na prvi procesni element i adresu ADR+2. Logično je da se ova naredba bavi izračunavanjem proizvoda A_2B_2 za narednu iteraciju, iako tekuća još nije završena. Dakle, operacija koja se dovodi na procesni element PE1 je MUL. Rezultat u prvom procesnom elementu biće upravo A_2B_2.
- Što se tiče drugog procesnog elementa, sada on treba da prenese ono D_1 koje je fiktivnim sabiranjem sa nulom, u prošlom koraku prenosio procesni element PE1. Dakle, operacija je ovde ADD a sadržaj lokalne memorije je 0.
- Treći procesni element, nema lokalnu memoriju. Jedan njegov operand je, kao što znamo, predodni rezultat predhodnog procesnog elementa, dakle, A_1*B_1/D_1. drugi operand je predhodni

rezultat njega samoga dakle prošli konačni rezultat (C_0). Dakle, rezultat trećeg procesnog elementa će biti A_1*B_1/D_1+C_0.

Četvrta mikronaredba:

- Na prvi procesni element se uvodi D_2 i operacija ADD, koja će sabrati nulu i D_2 u cilju njenog prenošenja dalje.
- Drugi procesni element deli (DIV) predhodni rezultat predhodnog procesnog elementa (A_2*B_2) sa operandom koji se nalazi na adresi ADR+2 u lokalnoj memoriji drugog procesnog elementa (D_2).
- Treći procesni elemet množi predhodni rezultat predhodnog procesnog elementa (a to je ono D_1 koje vučemo od početka, sabirajući ga sa nulom) sa rezultatom samog sebe (jer nema memoriju) iz prošlog kruga.
- Ovim smo konačno dobili ono što smo tražili a to je C_1.

Ovaj opis prve četri naredbe je više nego dovoljan za razumevanje ostatka mikroprograma. Zato neću dalje opisivati mikroprogram :)

ZADATAK BR. 17

Projektovati četvorostepeni protočni sistem za izračunavanje elemenata niza C na osnovu izraza:

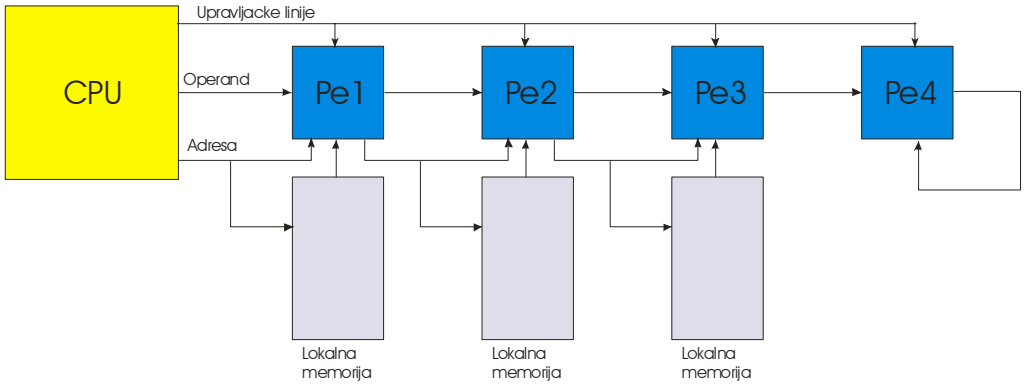
$$c_i = \frac{a_i^2 - b_i^2}{e_i^2 - f_i + g_i} = \frac{(a_i - b_i)\cdot(a_i + b_i)}{e_i^2 - f_i + g_i}$$

Svaki protočni element može izračunavati osnovne aritmetičke operacije i NOP naredbu. Na raspolaganju su procesni elementi sa i bez lokalne memorije.

Pretpostavlja se da kada Peocesni Element izvršava naredbu NOP da se operand sa njegovog ulaza preslikava za izlaz (bez obrade). Operandi i rezultati su 16 bitni brojevi, a lokalne memorije svih procesnigh elemenata su mapaciteta 64K 16bitnih reči. prilikom projektovanja sistema ne koristiti multipleksere.

Odrediti fotrmat mikronaredbe opisanog sistema i simbolički prikazati izvršenje dela mikroprogram, kao i sadržaj lokalnih memorija koji mu odgovaraju za izračunavnje prva tri elementa niza C.

Rešenje zadatka r. 17



Odmah ćemo pisati mikroprogram:

OP	ADR.	Ko1	Ko2	Ko3	Ko4	REZ1	REZ2	REZ3	REZ4	LM1	LM2	LM3
E_1	ADR	MUL	NOP	NOP	NOP	E_1^2	X	X	X	E_1	F_1	G_1
A_1	ADR+1	SUB	SUB	NOP	NOP	A_1-B_1	$E^2_1-F_1$	X	X	B_1	0	
A_1	ADR+2	ADD	ADD	ADD	NOP	A_1+B_1	A_1-B_1	$E^2_1-F_1+G_1$	X	B_1		
E_2	ADR+3	MUL	ADD	ADD	NOP	E_2^2	A_1+B_1	A_1-B_1	$E^2_1-F_1+G_1$			
A_2	ADR+4	SUB	SUB	ADD	DIV	A_2-B_2	$E^2_2-F_2$	A_1+B_1	$(A_1-B_1)/\ E^2_1-F_1+G_1$			
A_2	ADR+5	ADD	ADD	ADD	MUL	A_2+B_2	A_2-B_2	$E^2_2-F_2+G_2$	$(A_1+B_1)*(A_1-B_1)/\ E^2_1-F_1+G_1$			
E_3	ADR+6	MUL	ADD	ADD	NOP		A_2+B_2	A_2-B_2	$E^2_2-F_2+G_2$			
A_3	ADR+7	SUB	SUB	ADD	DIV			A_2+B_2	$(A_2-B_2)/\ E^2_2-F_2+G_2$			
A_3	ADR+8	ADD	ADD	ADD	MUL				$(A_2+B_2)*(A_2-B_2)/\ E^2_1-F_2+G_2$			
		NOP	ADD	ADD	NOP							
		NOP	NOP	ADD	DIV							
		NOP	NOP	NOP	MUL							

Tabela nije potpuno kompletna, ali je dovoljno da se uvidi osnovni princip.

PROTOČNO IZVRŠENJE INSTRUKCIJA

Protočnost je svuda prisutna pri izvršavanju instrukcija. Protočnost se tek pojavila u svetu računara prelaskom na RISC arhitekturu, zato što je upravo ta RISC arhitektura imala sve one karakteristike koje zahteva protočnost.

CISC Complex Instruction Set Computer

RISC Reduced Instruction Set Computer

Dugo je vladalo ubeđenje da CPU treba biti što složeniji. Smatralo se da će se time povećati performanse i znatno uprostiti pisanje kompilatora. Naravno, ovo je tačno iz ugla sekvencijalnog izvršavanja programa, ali iz ugla protočnosti cela ova priča pada u vodu. Zašto? Zato što se povećanjem složenosti CPU–a dobijaju jako složene naredbe koje dosta duže traju od prostih naredbi. Ovo je u suprotnosti sa osnovnim postulatom protočnosti koji govori o tome da operacije (instrukcije) moraju da traju približno isto!

Karakteristika CISC–a:

Ova arhitektura bila je aktivna 60–tih godina ovog veka. Tada se težilo da CPU bude što kompleksniji što bi za sobom povuklo lakše pisanje kompilatora. Naravno, za ovakav trend se ispostavilo da je pogrešan smer napredovanja jer se protočnost nikako nije mogla ostvariti s obzirom da se dužina instrukcija razlikovala čak u rasponu od 3–120 ciklusa takta. Dakle, ovo je prevaziđena tehnologija koju ćemo pomenuti samo informativno.

- Ova arhitektura ima veliki broj instrukcija (150–310)
- Instrukcije su različite dužine i kreću se od 1 do 11 reči
- Javlja se veliki broj različitih formata instrukcija
- U radu se koriste registri specijalne namene tj. mali je broj registara opšte namene koji su dostupni programeru.
- Veliki broj adresnih režima rada (18 režima kod motorole)
- Vreme izvršenja instrukcija je razičito i varira u velikim granicama.
- Mikroprocesorsko upravljanje (koje je zahtevalo veliku mikroprogramsku memoriju jer su postojale jako složene naredbe koje su zahtevale jako dug mikroprogram.
- Za svaku instrukciju se piše mikroprogram i smešta se u mikroprogramsku memoriju
- Osnovna f–ja kompilatora je da premošćava semantički procep koji postoji između mašinskih instrukcija i iskaza na višim programskim jezicima.
- Ako su mašinske instrukcije složenije i bliže višem programskom jeziku f–ja kompilatora je beznačajnija a samim tim se lakše i piše sam kompajler.

Kod CISC procesora mogu se identifikovati sledeće **etape** u izvršavalju instrukcija:

- Pribavljanje instrukcije IF
- Dekodiranje instrukcije ID
- Izračunavanje efektivne adrese operanda EA
- Pribavljanje operanda OF
- Izvršenje i upis rezultata EXE

Kod CISC računara kompilatori su izuzetno prosti. Osnovna funkcija kompilatora je da premosti semantički procep koji postoji između instrukcija na višim programskim jezicima i instrukcijama na mašinskom jeziku. Dakle, što su mašinske instrukcije složenije i bliže instrukcijama sa viših programskih jezika, to je pisanje kompilatora prostiji posao.

Brojnim ispitivanjima se dokazalo da se svega 25% instrukcija koristi skoro 95% vremena a onih 75% instrukcija se takoreći ne koristi. Te instrukcije, koje se nisu koristile su inače bile jako složene. To je navelo projektante da počnu da razmišljaju o tome da projektuju sistem koji će imati samo proste instrukcije. Tako je nastala RISC arhitektura.

Kod CISC procesora na upravljačku jedinicu otpada 40–60% površine čipa. Kod RISC procesora upravljačka jedinica zauzima 10% čipa, a preostali prostor iskorišćen je za povećanje broja registara opšte namene i za ugradnju on–chip keš memorije.

Karakteristike RISC–a

- Dijametralno suprotan CISC–u
- Svega 25% instrukcija CISC–a se koristi u 95% vremena (to su obično kratke naredbe, dok se one složenije javljaju samo u 5% vremena)
- Ovih 25% instrukcija se hardverski implementiraju a samim tim i brže izvršavaju dok se ostale, koje se ređe koriste, implementiraju softverski.
- Nemaju mikroprograme niti mikroprogramsku memoriju dakle, upravljačka jedinica je hardverska a ne softverska.
- Ukupan rezultat je na strani RISC–a a ne na strani CISC–a.
- Upravljačka jedinica zauzima 10% čipa (kod CISC–a između 40%–60%) a ostala površina se koristi za druge namene.
- 1986 je izašao prvi RISC računar.
- Mali broj instrukcija (manji od 100)
- Mali broj različitih formata instrukcija (najviše 3)
- Sve instrukcije su iste dužine (32 bita)
- Vreme izvršavanja je isto
- Mali broj adresnih režima (1 ili 2)
- Veliki broj registara opšte namene a mali broj registara specijalne namene.
- Jedine naredbe koje se obraćaju memoriji su LOAD i STORE a sve ostale su registarsko–registarske.
- Upravljanje je hardversko a ne mikroprocesorsko.
- JAKO IZRAŽENA PROTOČNOST
- Kompilatori su veoma moćni i imaju ključnu ulogu u postizanju performansi
    - Minimiziraju obraćanje memoriji
    - Maksimizuju upotrebu registara opšte namene
    - Postoji algoritam za dodelu registarskih promenljivih

Kompilatori su u RISC arhitekturi dobili važnu ulogu jer je od njihovog kvaliteta zavisila brzina izvršavanja programa. Kompilator je taj koji je trebao da minimizuje broj obraćanja memoriji a da maksimizuje broj obraćanja registrima opšte namene. Za to se koristi matematička teorija **bojenja grafa**, ili neki **heuristički metodi**. Ukoliko bi se koristili heuristički metodi minimalan broj registara bi trebao da bude 16. Ovaj uslov je samo nekada predstavljao problem, danas svi računari imaju preko 16 registara. Sve u svemu RISC arhitektura je dijametralno suprotna CISC arhitekturi.

Prikaz arhitekture RISC procesora

U arhitekturi RISC procesora možemo prepoznati **upravljačku jedinicu** i **puteve podataka** (to je ono što se ranije zvalo **aritmetička jedinica**).

Data path (putevi podataka) se sastoji od:

- ALU
- Registarskog fajla (veliki broj registara opšte namene). Na njegovom izlazu su lečevi A i B a na njegovom ulazu je leč C (Registri za čitanje i upisivanje) sadržaja u registre.
- Registri specijalne namene PC (program counter), MAR (memory addres register) tu se čuva adresa memorijske adrese kojoj se pristupa. IR (instruction register), MDR (memory data register prihvata podatak koji se čita iz memorije pre nego što se on upiše u registar fajl)

Memorija iz koje se pribavljaju instrukcije (sve instrukcije su 32–bitne). Postoji 3 različita formata instrukcije. I, R i J format. O ovim formatima možemo reći sledeće.

I format

OP	RS1	RD	Immed

OP zauzima 6 bitova

RS1 zauzima 5 bitova

RD zauzima 5 bitova

Immed zauzima 16 bitova

Ovaj format imaju instrukcije:

- Load
- Store
- Branch
- ALU instrukcije koje koriste neposredni operand a drugi operand je iz registra opšte namene.

Polja u ovom formatu su sledeća:

OP Je operacioni kod operacije

PS1 U slučaju LOAD i STORE instrukcija to je polje gde je zapamćena bazna memorijska adresa;

U slučaju BRANCH instrukcije to je polje registra čiji se sadržaj testira;

U slučaju ALU instrukcije to je polje jednog operanda (izvorišnog).

RD U slučaju LOAD instrukcije to je polje odredišnog registra. U njega se vrši upis

U slučaju STORE instrukcije to je polje registra čiji se sadržaj pamti u memoriju

U slučaju BRANCH instrukcije RD je uvek jednak nuli

U slučaju ALU instrukcije to je polje odredišnog registra

Immed U slučaju LOAD/STORE instrukcije to je offset koji se dodaje baznoj adresi koja se nalazi u RS,

U slučaju BRANCH instrukcije to je adresa skoka

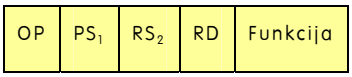
U slučaju ALU instrukcije to je drugi (neposredni) operand.

Primer:

LW	R1,30(R2);	R1 <– MEM(30+(R2))
SW	500(R4),R3	MEM(500+(R4)) <– (R3)
ADDI	R1,R2,#3	R1 <– (R2)+2
BEQZ	R1,ime	IF(R1=0) then PC <– PC + ime

R format

Imaju ga ALU instrukcije sa operandima u registrima (oba) opšte namene.



OP zajedno sa Funkcija definiše o kojoj se operaciji radi.

RS1 je registar prvog izvorišnog operanda

RS2 je registar drugog izvorišnog operanda

RD je registar odredišnog operanda

Dejstvo ovih instrukcija je:

RD <– (RS1) op (RS2)

Na primer:

MUL R1,R2,R3; R1 <– (R2)*(R3)

J format

Se koristi za instrukcije bezuslovnog skoka. Adresa skoka se računa tako što se offset dodaje sadržaju programskog brojača.

OP Je operacioni kod instrukcije.

Offset Se dodaje PC–u (programskom brojaču)

Na primer:

JMP ime ;PC <– PC+ime

Faze u izvršavanju instrukcije

Kod RISC procesora mogu se identifikovati sledeće **faze** u izvršavanju instrukcije:

1. Faza pribavljanja instrukcije — IF (instruction Feach)
2. Faza dekodiranja instrukcije i pribavljanja operanda — ID (instruction decode)
3. Izvršavanje instrukcije — EX (execution)
4. Pristup memoriji — MEM
5. Upis rezultata u registarski fajl — WB (write back)

Faza pribavljanja instrukcije IF:

Na osnovu sadržaja PC–a se pristupa MEM, pribavlja se instrukcija i smešta u IR.

IR <–MEM(PC)

Vrši se inkrementiranje ili povećanje PC–a za 4 zato što su instrukcije 32 bitne a znamo da je 32 bita = bajta tj. 4 reči. Tako uvećan PC se pamti u privremenom registru

NPC <– PC+4

Razlog ovome je što neznamo da li idemo na sledeću naredbu ili skačemo negde pod dejstvom naredbe JMP ili bilo koje BRANCH naredbe. U zavisnosti od te odluke PC dobija vrednost NPC–a ili pak adresu naredbe grananja.

Faza dekodiranja instrukcije i pribavljanja operanda:

Paralelno se obavljaju obe operacije. Tj. paralelno se odvija dekodiranje instrukcije i pribavljanje operanada. To je moguće zato što se polja operanada u formatu instrukcije uvek nalaze na fiksim mestima.

A <– reg (IR6.....IR10) Pristupamo registarskom fajlu gde se broj registara nalazi u polju IR 6...10

B <– reg (IR11....IR15) Pristupamo registarskom fajlu,gde se broj registara nalazi u polju IR 11...16

Imm <– $(IR16)^{16}$ ## (IR16 .... IR32)

U ovoj fazi se paralelno odvija **dekodiranje** instrukcije i **pribavljanje** operanda. Ovo je moguće zato što se polja operanada u formatu instrukcije uvek nalaze na fiksnim mestima. Operandi se pribavljaju iz registara opšte namene, a za njihov smeštaj se koriste privremeni registri A,B i Imm. U toku pribavljanja operanada još uvek nije poznato o kojoj se instrukciji radi, niti o kom formatu. Ipak ne može doći do greške. Najlošije što se može dogoditi je da se pribavi neki operand višak, a od viška glava ne boli. Loše bi bilo kada bi bilo manjka.

Izvršavanje instrukcije:

Razlikuje se za različite instrukcije. Ako je dekodirana ALU instrukcija (sa neposrednim operandom ili sa oba operanda u registrima opšte namene) rezultat se smešta u privremeni registar koji se zove ALU(OutPut).

U zavisnosti od toga koja je naredba dekodirana dešavaće se sledeće:

- ALU sa neposrednim operandom: ALU(Output) <– A op Imm

  ALU(Output) je privremeni registar u koji se smešta rezultat izračunavanja ALU instrukcije.

- ALU sa operandima u registrima opšte namene: ALU(Output) <– A op B (R format)

- Naredbe za obraćanje memoriji LOAD/STORE: izračunava se efektivna adresa na osnovu koje se u sledećoj fazi pristupa memoriji.

  ALU(Output) <– A + Imm

- Naredbe grananja: vrši se testiranje uslova i izračunavanje adrese grananja.

  ALU(Output) <– NPC + Imm

  Cond <– A op): uvek se testira na nulu Cond je privremeni registar u koga se smešta rezultat testiranja.

  Op može biti =, <,>,<=,>= (BEQZ,BNEQZ). O kom je operandu reč definiše se kodom same instrukcije.

Dakle, u ovoj fazi ALU se koristi za aritmetičko logičke operacije i izračunavanje EA (efektivne adrese). Obe ove funkcije aritmetičko logičke jedinice je moguće kombinovati ujednom stepenu zato što ni jedna instrukcija ne zahteva istovremeno izračunavanje EA ili ALU operaciju.

Pristup memoriji:

Ovo je jedina faza u kojoj se vrši obraćanje memoriji. Instrukcije u ovomciklusu su LOAD/STORE/BRANCH. Tek na kraju ove faze se zna rezultat grananja (adresa sledeće instrukcije). Aktivnosti koje se obavljaju u ovoj fazi su različite u odnosu na tip instrukcije.

- STORE naredba upisuje sadržaj odredišnog reg u memorijsku lokaciju po adresi koja je izračunata u prethodnoj fazi. MEM (ALU(Output)) <– B

- LOAD naredba pristupa memoriji po adresi izračunatoj u prethodnoj fazi i sadržaj se upisuje u LMD (Load Memory data):

  LMD <– MEM(ALU(Output)) LMD je ovde privremeni registar.

- BRANCH naredba menja sadržaj PC–a u zavisnosti od ishoda komparacije dakle,

  IF cond THEN PC <– ALU(Output)

  ELSE PC <– NPC

  Instrukcije STORE i BRANCH su kompletirane po završetku ove faze.

Upis rezultata u registarski fajl:

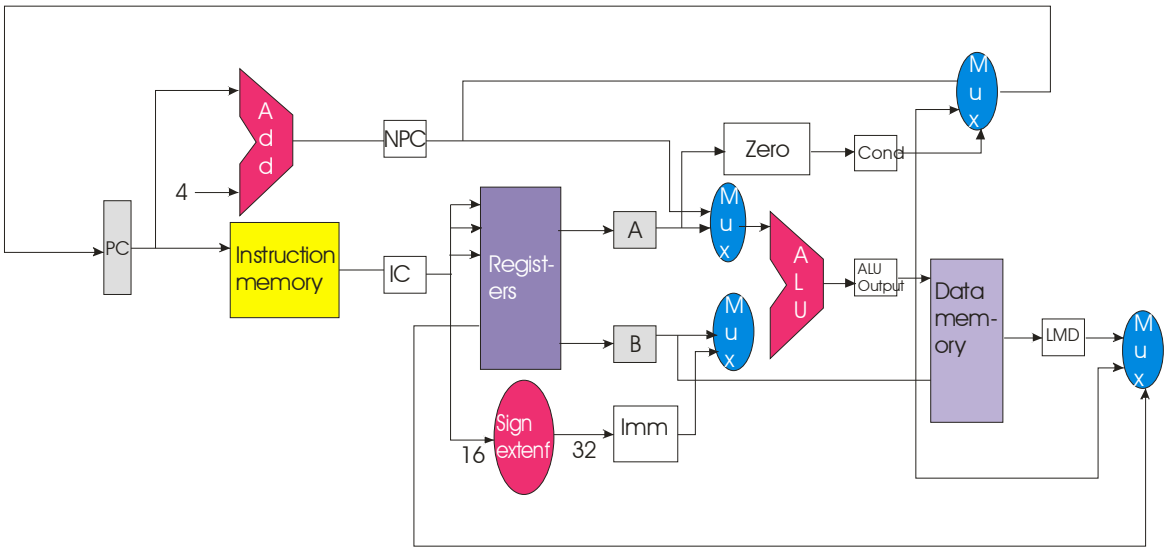
Ovo je jedina faza u kojoj se vrši upis u registre opšte namene. Naravno, u zavisnosti od instrukcije i formata naredbe zavise i faze.

REG (IR11.....IR15) <– ALU(Output) :Alu instrukcija tipa Neposredni operand – Registar (I format)

REG (IR16.....IR20) <– ALU(Output) :ALU instrukcija tipa registar – registar (R format)

REG (IR11.....IR15) <– LMD :LOAD instrukcija. (sadržaj LMD–a se upisuje u registar.

Uvođenje protočnosti



Protočnost se uvodi tako što se u svakom kloku inicira po jedna instrukcija.

Uvođenjem protočnosti se otvaraju neki problmi, od kojih ćemo pomenuti najveće:

- Sadržaj PC–a se mora inkrementirati u svakom klok ciklusu dakle, mora biti poznat u IF fazi), ali se on zna tek na kraju MEM faze zar ne ? Problem predstavljaju instrukcije koje mogu da promene sadržaj PC–a. Trebalo bi postignuti da se novi sadržaj PC–a zna već na kraju IF faze.
- Osim toga, da bi u svakom ciklusu takta pribavljali novu operaciju treba pristrupiti memoriji, zato se uvode keš memorije jer čitanje iz memorije traje 10–20 ciklusa takta. MEM faza jedne i IF faza druge naredbe se međusobno mogu preklopiti. Upravo zbog toga su odvojene keš memorije za podatke i keš memorija za instrukcije. Dakle, postoje i posebne keš memorije za podatke i keš memorije za instrukcije.

U toku protočnog izvršavanja instrukcije koriste se neki privremeni registri. Ovi registri se moraju uvesti kako bi se sačuvali svi podaci neophodni za pravilno izvršavanje instrukcije. Oni se zovu **protočni lečevi**. Kako instrukcija napreduje kroz protočne stepene, tako se informacije prenose kroz protočne lečeve. Dakle, protočnost se ne bi mogla realizovati samo pomoću onih lečeva koje imamo kod neprotočnog sistema, jer bi se rezultat izgubio pre nego što bi bio potreban.

Do većih problema u protočnim sistemima dolazi kad se jave instrukcije koje ne traju jednako. Onda se dozvoljava takozvano **vanredosledno izvršavanje**, a to za sobom povlači nove probleme. Aktivnosti koje se obavljaju u protočnom sistemu iste su kao i aktivnosti koje se obavljaju u neprotočnom sistemu (samo su pridodati nizovi protočnih lečeva u kojie se smeštaju podaci. Protočni registri imaju polja: IR, NPC, ALU(Output)....tako da se tu smeštaju svi međurezultati bitni za dalje izvršenje naredbe.

Stage	Any instruction		
IF	IF/ID.IR <–MEM(PC) IF/ID.NPC.PC <–(if EX/MEM.cond (EX/MEM.NPC) else (PC+4));		
ID	ID/EX.A <–Regs(IF/ID.IR_6...IR_{10}); ID/EX.B <–Regs(IF/ID.IR_{11}...IR_{15}); ID/EX.NPC <–IF/ID.NPC; ID/EX.IR <–IF/ID.IR; ID/EX.Imm <–(IR_{16})^{16}##IR_{16}.....IR_{31};		
	ALU instruction	Load/Store instruction	Branch instruction
EX	EX/MEM.IR <– ID/EX.IR; EX/MEM.ALU_{OutPut} <– ID/EX.A op ID/EX.B; OR EM/MEM.ALU_{OutPut} <– ID/EX.A op ID/EX.Imm; EX/MEM.cond <–0;	EX/MEM.IR <– ID/EX.IR EX/MEM.ALU_{OutPut} <– ID/EX.A + ID/EX.Imm; EX/MEM.cond <– 0; EX/MEM.B <– ID/EX.B	EX/MEM.ALU_{OutPuT}<– ID/EX.NPC+ID.EX.Imm; EX/MEM.cond <– (ID/EX.A op 0);
MEM	MEM/WB.IR <–EX/MEM.IR; MEM/WB.ALU_{OutPut} <– EX/MEM.ALU_{OutPut};	MEM/WB.IR <– EX/MEM.IR; MEM/WB.LND <– Mem(EX/MEM.ALU_{OutPut}); OR Mem(EX/MEM.ALU_{OutPut}) <– EX/MEM.B;	If (cond) PC
WB	Regs(MEM/WB.IR_{16}..IR_{20}) <– MEM/WB.ALU_{OutPut}; OR Regs(MEM/WB.IR_{11}...IR_{15}) <– MEM/WB.ALUOutPut;	Regs(MEM/WB.IR_{11}...IR_{15}) <– MEM/WB.LMD;	

Hazardi

U **idealnom** slučaju bi performanse protočnog sistema bile 1 instrukcija po jednom ciklusu takta, ali postoji niz problema koje zanivamo **HAZARDIMA**.

1. Jednu grupu problema izražavaju instrukcije koje mogu da promene sadržaj PC–a (JMP, BRANCH, CALL, RETURN). Ovi hazardi nazivaju se **kontrolni hazardi**.
2. Zavisnost po podacima **hazardi podataka** nastaju zato što je redosled pristupa operandima izmenjen uvođenjem protočnosti u odnosu na sekvencijalno izvršenje instrukcija.
3. Strukturalne zavisnosti **strukturalni hazardi** nastaju usled toga što se u različitim fazama izvršenja instrukcije zahteva korišćenje istog hardvera (hardverskog resursa). Do ovih zavisnosti dolazi samo ako protočnost nije dovoljno duboka.

Hazardi po podacima

Primer: Format ovih instrukcija je:

Operacija	RD	RS1	RS2

ADD R1,R2,R3

SUB R4,R1,R5

AND R6,R1,R4

OR R7,R1,R9

XOR R10,R1,R11

U ovom protočnom sistemu sve hazarde je moguće detektovati u **ID** fazi. To ćemo uraditi tako što poredimo odredišni operand neke prethodne i izvorni operand tekuće instrukcije. Ako se detektuje hazard, zaustavlja se protočni sistem. To bi dosta usporilo sistem pa se koristi **premošćavanje.**

Premošćavanje se izvodi tako što se, na primer, rezultat iz EX/MEM.ALU(Output)–a prosleđuje ponovo na ulaz ALU–a. Ovim premošćavanjem smo rešili problem zavisnosti po podacima zar ne ? Premošćavanjem (*bypassing, frowarding*) se može rešiti i problem zavisnosti po podacima između dve instrukcije koje su susedne (kao što smo malopre objasnili) ili zavosnost popodacima između prve i treće naredbe tako što se iz ME.WB.ALU(Output)–a prosledi na ulaz Alu–a ali za ovo je neophodno uvesti dodatne multipleksere na ulaz ALU–a.

Osim toga, vidimo da u navedenom primeru instrukcije ADD i OR u istom ciklusu takta pristupaju registrima. ADD radi upisa a OR radi čitanja operanda. Pošto apsoluno istovremeni pristup ne može da se vrši, ovaj problem se rešava tako što se u prvoj polovini periode takta vrši upis u registar a u drugoj polovini takta čitanje iz registra (read signal je aktiviran u drugoj polovini peride clock–a). Jedino što se dodaje je hardver za otkrivanje haarda (za kompariranje odgovarajućih polja u IR) i multiplekseri na ulazu ALU–a kojima upravlja upravo hardver za detektovanje hazarda.

Hazardi podataka su najčešći tip hazarda. Dobili su nazive po redosledu pristupa operandima (podacima) koji mora biti sačuvan uvođenjem protočnosti.

Primer: Posmatramo instrukciju I i J gde je I < J tj. I predhodi J

1. Do hazarda može doći ako J pokušava da pročita operand pre nego što je I izvršila upis u odgovarajući registar. To je **RAW** hazard. Oni se lako rešavaju premošćavanjem.

2. Do **WAR** hazard–a (Write After Read) dolazi ako J pokuša da izvrši upis u odredišni registar a da I još uvek nije obavila čitanje iz tog registra. Ovaj hazard se neće nikada desiti za naš primer, pošto je upis na kraju, ali za **Multycycle** instrukcije i za neredosledno izvršavanje može doći do ovog hazarda. Do ovog hazarda češće dolazi kod CISC procesora npr: kod naredbi autoinkrementiranja i autodekrementiranja.

Na primer:

IF	ID	EA	OF	EX

EA Efektivna adresa

OF Operand feach

Primer:

1. ADD (R1),R1,R3
2. ADD R4,+(R1),RF

IF	ID	EA	OF	EX	
	IF	ID	EA	OF	EX

U naredbi numerisanoj kao 1. u EA fazi se ustanovljava da je u R1 adresa. Međutim u naredbi u drugom redu se R1 inkrementira. Tako da tu nastaje hazard. Na slici su mesta na kojima nastaju hazardi obojena crvenom bojom.

3. **WAW** hazard (**W**rite **A**fter **W**rite) do njega dolazi kada J pokuša da upiše vrednost u odredišni registar pre upisivanja od strane instrukcije I. Na taj način se bespovratno gubi sadržaj registra.

Primer:

1. ADD R1,R2,R3
2. SUB R1,R4,R5

Ovaj hazard nastaje ukoliko prvo naredba pod rednim brojem 2. prvo upiše sadržaj u registrar R1 pa tek onda naredba pod rdnim brojem 1. U tom slučaju se sadržaj registra trajno gubi a dalje izvršavanje programa postaje nekorektno.

U našem protočnom sistemu ne može doći do ovakvih hazarda, ali može u sistemima gde postoji mogućnost upisa u više hardverskih stepena, ili ako je dozvoljeno neredosledno izvršavanje instrukcija, ili ako je EX faza različite duzine za različite instrukcije.

Primer:

IF	ID	EX	ME	WB		
	IF	ID		EX	ME	WB
		IF		ID	EX	ME

LW R1,31(R6)

```
ADD     R4,R1,R7
SUB     R5,R1,R8
AND     R6,R1,R7
```

Moguće je i sledeće premošćavanje: Da se izlazi LMD–a iz MEM/WB leča vode na MUX–exe ispred ALU–a. Bez ovog premošćavanja, ukoliko bi se odmah nakon LW u programu našla naredba koja koristi pročitani podataka (podatak koji je malopre pročitala LW naredba) imali bi smo zastoj od 2 ciklusa takta. Sa premošćavanjem, u ovom slučaju imamo zastoj od samo jednog ciklusa takta.

Hazardi se otkrivaju još u ID fazi. Za to se koristi poseban **hardver za otkrivanje hazarda** koji za sve susedne instrukcije poredi odgovarajuća polja operanada iz registra instrukicija i na osnovu rezultata tog poređenja on upravlja radom multipleksera koji su uvedeni zbog premošćavanja. Za LOAD instrukciju se vrši poređenje sa dve naredne instrukcije. Ukoliko premošćavanje ne može da se izvrši (ako su zavisne LOAD i sledeća instrukcija) utvrđujemo postojanje zavisnosti i protočni sistem se zaustavvlja. Multiplekseri moraju imati odgovarajući broj ulaza kako bi mogla da se izvrše sva ova premošćavanja. Da bi se u ID fazi detektovao hazard i da bi se aktivirali odgovarajući ulazi u multipleksere neophodno je da se obavi 10 različitih poređenja.

U MUX–ima ima 3 ulaza:

1. Za prosleđeni iz EX/MEM (za ALU instrukcije)
2. Odgovara prosleđivanju iz MEM faze
3. Odgovara premošćenju iz MEM/WB faze.

Prvih 8 redova u tablici odnosi se na RAW hazarde uzrokovane ALU instrukcijama, a poslednja 2 reda odnose se na RAW hazarde uzrokovane LOAD instrukcijama.

```
MUL     R1,R9,R10
LW      R1,31(R6)
```

Zaustavljanje protočnog sistema, ako imamo hazard uzrokovan LOAD instrukcijom ne može da se izbegne dok ne protekne MEM faza load instrukcije. U ovom slučaju zaustavljanje protočnog sistema se izbegava korišćenjem kompilatora koji izbegava generisanje programskog koda kod koga iz LOAD sledi instrukcija koja koristi pročitani podatak. Ovo preuređenje koda izvodi kompilator u fazi prevođenja programa, a sama tehnika se naziva **Zakasneli LOAD**. To je u suštini kompilatorska tehnika. Ova tehnika zahteva veliki broj registara opšte namene, što je ispunjeno kod RISC procesora.

Primer:

Napisati kod koji će izbeći zaustavljanje zbog RAW hazarda za sledeći niz naredbi: a=b + c; d=e – f;

```
LW      Pb, b
LW      Rc, c
LW      Re, e
ADD     Re, e
LW      Rf, f
```

SW	Rf, f
SW	a, Ra
SUB	Rd, Re, Rf
SW	Di, Rd

U nastavku teksta data je tabela malopre pomenutih ispitivanja da li postoji hazarda po podacima ili ne!

Protočni Registar Izvorišne instrukcije	Tip instrukcije	Protočni Registar Odredišne instrukcije	Instrukcija Koja je u ID fazi	Gde se Prosleđuje Premošćeni rezultat	Koje Testiranje Treba obaviti
EX/MEM	Register–register ALU	ID/EX	Reg.–reg. ALU, ALU Imm Load, Store, Branch	Top ALU Input	EX/MEM.IR_{16}..IR_{20} =ID/EX.IR_6...IR_{10}
EX/MEM	Register–register ALU	ID/EX	Register–register ALU	Bottom ALU input	EX/MEM.IR_{16}..IR_{20} =ID/EX.IR_{11}..IR_{15}
MEM/WB	Register–register ALU	ID/EX	Reg.–reg. ALU, ALU Imm Load, Store, Branch	Top ALU Input	MEM/WB.IR_{16}..IR_{20} =ID/EX.IR_6..IR_{10}
MWM/WB	Register–register ALU	ID/EX	Register–register ALU	Bottom ALU input	MEM/WB.IR_{16}..IR_{20} ID/EX IR_{11}..IR_{15}
EX/MEM	ALU Imm	ID/EX	Reg.–reg. ALU, ALU Imm Load, Store, Branch	Top ALU Input	EX/MEM.IR_{11}..IR_{15} =ID/EX.IR_6...IR_{10}
EX/MEM	ALU Imm	ID/EX	Register–register ALU	Bottom ALU input	EX/MEM.IR_{11}..IR_{15} =ID/EX.IR_{11}...IR_{15}
MEM/WB	ALU Imm	ID/EX	Reg.–reg. ALU, ALU Imm Load, Store, Branch	Top ALU Input	MEM/WB.IR_{11}..IR_{15} =ID/EX.IR_6..IR_{10}
MWM/WB	ALU Imm	ID/EX	Register–register ALU	Bottom ALU input	MEM/WB.IR_{11}..IR_{15} =ID/EX.IR_{11}..IR_{15}
MEM/WB	LOAD	ID/EX	Reg.–reg. ALU, ALU Imm Load, Store, Branch	Top ALU Input	MEM/WB.IR_{11}..IR_{15} =ID/EX.IR_6..IR_{10}
MWM/WB	LOAD	ID/EX	Register–register ALU	Bottom ALU input	MEM/WB.IR_{11}..IR_{15} =ID/EX.IR_{11}..IR_{15}

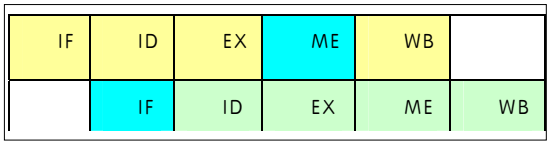
Kontrolni hazardi

Ovi hazardi su uzrokovani instrukcijama koje mogu da promene sadržaj programskog brojača. To su naredbe BRANCH, JUMP, CALL, RETURN. Uzmimo na primer: BRANCH instrukciju. To je naredba sa jednim uslovom za testiranje.

Primer:

BRANCH

I+1



Zanimljivo je napomenuti da se tek u MEM fazi BRANCH naredbe zna adresa skoka (novi PC) a da je tek posle IF faze u narednoj naredbi moguće zaustavljanje zato što se do tada ne zna da li je to BRANCH naredba. Postavlja se pitanje šta raditi sa instrukcijom koja sekvenciojalno sledi iza naredbe grananja (da li pretpostaviti da se granabnje obavilo ili ne ?)

Moguće je zaustaviti protočni sistem i čekati da se grananje okonča, i tek tada kad se zna ishod nastaviti u skladu sa rezultatom grananja. Tada se gubi 3 ciklusa takta.

Negde oko 10–20% naredbi su naredbe grananja a ovde je gubitak 3 ciklusa takta. Što ranije treba utvrditi da li je do grananja došlo ili ne i za slučaj da do grananja dolazi što ranije odrediti novi sadržaj PC–a. Ovo je sve u duhu smanjenja stepena degenerisanja performans protočnog sistema usled nailaska na naredbu grananja.

Sve ovo treba utvrditi što ranije u protočnom sistemu. Sa našom pretpostavkom o jednostavnosti instrukcije grananja ovo se može završiti već u ID fazi. Da podsetim, pretpostavili smo da se naredba grananja sastoji samo od ispitivanja sadržaja nekog registra na nulu. (kada već pristupamo registarskom fajlu, moguće je izvršiti i kompariranje, što zahteva postojanje **komparatora** u ID stepnu.

U ID fazi se takođe vrši izdvajanje neposrednog operanda iz instrukcije, pa je u ID fazi moguće izvršiti i sabiranje kako bi se dobila adresa skoka, za šta je u ID stepen potrebno dodati jedan **sabirač** (jer se ALU koristi za neke druge operacije u fazama predhodnih instrukcija). Ovim su aktivnosti u ID fazi modifikovane i obavljaju se za sve instrukcije jer se još uvek ne zna da li se radi o naredbi grananja ili ne. tek na kraju ID faze, kada se instrukcija dekodira, možemo postaviti novi sadržaj u programski brojač.

Faze:

IF	IF/ID.IR	<– MEM(PC)
	IF/ID.NPC	PC <– PC+4
ID	ID/EX.A	<– IF/ID IR0.....IR10
	ID/EX.B	<– IF/ID IR11.....IR15
	ID/EX.Imm	<– $(IR16)^{16}$##IR16...IR31
	ID/EX.NPC	<– IF/IP.NPC + Imm
	COND	<– ID/EX A opB

	IF COND THEN PC <– NPC

Predhodni primer će sada izgledati ovako.

IF	ID	EX	ME	WB		
		IF	ID	EX	ME	WB

Dakle, gubi se samo jedan ciklus takta. Čak i ovaj ciklus takta je moguće izbeći u nekim slučajevima. NPR: čim se dekodira instrukcija grananja da se krene sa pribavljene i ciljne adrese (**pretpostavlja** se da će do grananja doći). to je dobro kada se čeka duže vreme na razrešenje uslova grananja. U našem (malom) primeru to nema efekta.

Druga mogućnost je da se **pretpostavi** da do grananja neće doći pa se tada pribavlja sledeća instrukcija, a ako se posle ispitivanja ispostavi da je do grananja ipak doslo, ta instrukcija se poništava.

Može se koristiti i *zakašnjeno grananje* (kompilatorska tehnika) preuređenjem programa. Sve naredbe koje se nalaze iza BRANCH naredbe se nalaze u takozvanom **prozoru zakašnjenog grananja**. U taj prozor treba postaviti instrukcije koje će se izvršavati bez obzira na ishod grananja.

Kolika je veličlna prozora? U ovom zadnjem primeru treba nam 1 blok ciklusa dakle, veličina prozora je 1. (jednu instrukciju treba postaviti u taj prozor da ne bi došlo do zastoja). Veličina prozora zavisi od dužine trajanja kontrolnog hazarda i od dubine protočnog sistema.

Primer:

ADD	R1,R2,R3
IF	R2=0 THEN

Ovo programce se moze preurediti tako sto se naredba ADD prebaci u prozor za kašnjenje koji je uznačen plavom bojom.

Dakle, dobilo bi se nešto ovako!

IF	R2=0 THEN
ADD	R1,R2,R3

Međutim, nije obavezno da se prozor za kašnjenje popuni!

Ako ne možemo da nadjemo takvu instrukciju, ne moramo forsirati preuređenje programa. To će kompilator uraditi za nas. Sve u svemu, ne smemo preuređenjem programa promeniti smisao programa tj. promeniti njegovu prvobitnu namenu.

Kod petlji pretpostavljamo da će do grananja doći pa se u slot stavlja instrukcija sa ciljne adrese grananja. **Procenat pogodaka** je oko 90%. Ova premeštanja su moguća samo ako se ne narušava korektnost programa. NPR: neka naredba se nesme izvršiti više puta nego što treba.

Primer:

ADD	R1,R2,R3
IF	R1=0 THEN
SUB	R4,R5,R6

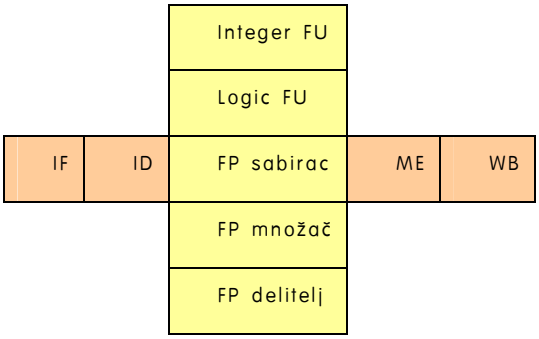
Ako pretpostavimo da do grananja ne dolazi onda program preuređujemo na sledeći način.

ADD	R1,R2,R3
IF	R1=0 THEN
SUB	R4,R5,R6

Ako su u pitanju petlje uvek se pretpostavlja da se grananje obavlja, pa se u prozor za grananje postavlja instrukcija sa puta obavljenog grananja, što zahteva da se modifikuje adresa skoka i da se instrukcija koja se ubacuje u prozor iskopira na više mesta u programu, kako bi se sigurno izvršila u svim situacijama. Ovo je moguće samo ako njen odredišni registar nije u isto vreme izvorišni registar naredne instrukcije sa puta neobavljenog grananja.

Protočni sistem sa više funkcionalnih jedinica

Ako imamo instrukcije koje rade sa podacima u pokretnom zarezu, imamo različite instrukcije koje različito traju i obično imamo različite funkcionalne jedinice za izvršavanje tih instrukcija pa možemo modifikovati naš početni protočni sistem. Ovako se donekle uvodi paralelizam u obradu.



Sada, posle **RAW** hazarda mogu da se jave i druge vrste hazarda po podacima (**WAW**). Još uvek ne može doći do **WAR** hazrda ako se čitanje vrši rano.

Primer:

DIVF	F0,F2,F10
SUBF	F0,F8,F10

F# su registri koji pamte podatke u dvostrukoj tačnosti tj. Floating Point vrednosti a ne integer kao R# registri. Mesto na kome se javio WAW hazard je označen plavom bojom. Tj. ta mesta su F0 u prvoj naredbi i F0 u drugoj naredbi.

Postoje odvojene funkcionalne jedinice za FDIV i FSUB. Ove instrukcije se mogu naći istovremeno u EXE fazi. EXE faza za SUB traje mnogo kraće nego za DIV pa se sub naredba ranije izvršava i upisuje u F0, pa tek onda DIV upiše u F0 i rezultat sub–a se gubi.

Ako bi između ove dve instrukcije bila neka instrukcija koja koristi F0 javio bi se **RAW** hazard i zastoji pa ne bi došlo do WAW hazarda. Do **WAW** hazarda može doći samo ako je neka instrukcija beskorisna (tj. ako se njen rezultat nikada ne zahteva. Ako ni se pak zahtevao došlo bi do RAW hazarda, međutim nijedan kompajler neće da generiše ovakvu instrukciju. Ipak, kod zakašnjenog grananja može da se javi ako se u slot stavi DIVF.

Primer za WAW hazard iako nije u pitanju beskorisna instrukcija:

BNEZ	R2,F00
DIVF	R0,F2,F4
.	
.	
SUBF	F0,F8,F4

Naredba koja je obelezena plavom bojom (DIVF) se nalazi u slotu za kašnjenje pod pretpostavkom da se grananje ne obavlja. Pošto je DIVF duža od SUBF ona će posle da upiše rezultat u F0 preko rezultata SUBF instrukcije.

Za rešavanje ovog problema postoje dve metode.

1. Kada se detektuje hazard, obično u ID fazi, treba da se zabrani izdavanje SUBF dok DIVF ne uđe u MEM fazu.
2. Da se kada se detektuje hazard SUBF–u ne dozvoljava izvršenje a DIVF–u se zabani upis u FR (registrer File) (sada ona nije beskorisna)

Još jedan problem je vezan za prekide. Kod instrukcija sa dugim vremenom izvršenja postoji i problem kod prihvatanja prekida.

DIVF	F0,F2,F4
ADD	F10,F10,F8
SUBF	F12,F12,F14

Ovo postaje opasno kada: U toku izvršenja DIVF instrukcije dođe do prekida, a ADDF je izvršila upis u RF. Nakon prekida ponovo se izvršava DIVF i ADDF kome je promenjen izvorišni operand F10. Dakle, problem je očigledan jer u F10 imamo pogrešan rezultat.

I za ovaj problem postoji rešenje!

Korišćenje **future file**–a. Rezultati svih instrukcija koje su izvršene van redosleda se smeštaju u bafer a upis u RF ide po redosledu pribavljanja instrukcija. Ovde je problem što bafer može biti vrlo veliki, a za svaku instrukciju koja se izvršava van redosleda treba pored mesta u baferu obezbediti i linije za premošćavanje do svih funkcionalnih jedinica. To uslovljava veliki obim hardvera.

Alternativna metoda je da za svaku instrukciju koja je ušla u izvršavanje a nije okončana se pamti dovoljno informacija tako da se za slučaj prekida mogu bezbedno izvorišni registri rekonstruisati. Ova metoda se zove **history file**. Ovde se ponovo radi o smeštanju podataka u neki bafer.

Koja od ove dve tehnike će biti implementirana zavisi od toga koji je set instrukcija računara i kolika su vremena izvršavanja tih instrukcija.

Napredne tehnike kod protočnih sistema

Uvođenjem protočnosti se mogu drastično povećati performanse procesora. U idealnom slučaju izvršava se jedna instrukcija u svakom ciklusu taktnog signala. Realno, zbog pojave *hazarda* imamo da se izveršava jedna instrukcija na svakih 1.7 ciklusa takta u proseku.

Da bi se performanse još poboljšale potrebno je koristiti **ILP** (**I**nstruction **L**evel **P**aralelism) tj. **paralelizam na nivou instrukcije**. Treba utvrditi koje bi instrukcije teorijski mogle izvršavati istovremeno.

Primer:

a)

LW	R1,30(R2)
ADDI	R3,R3,#1
ADD	R4,R5,R5

b)

ADDI	R3,R3,#1
ADD	R4,R3R2
MUL	R5,R3,R4

U primeru pod a) ni jedna instrukcija ne zavisi od neke druge, sve tri bi mogle da se izvršavaju istovremeno zar ne ? Zaključujemo da je ILP=3.

U primeru pod b) ADD zavisi od ADDI, a MUL zavisi od ADD. Ovde ne može biti nikakvog paralelizma na nivou instrukcije. Instrukcije moraju da se izvršavaju strogo sekvencijalno dakle, ILP=1.

a) ILP=3 zato što su sve tri instrukcije međusobno nezavisne i mogu se istovremeno izvršavati.

b) Nema nikakvog paralelizma ILP=1

Postavlja se pitanje kako povećati ILP ?

Odgovor: Paralelizam koji je rasopoloživ u osnovnom bloku instrukcija (kodni segment koji ne sadrži naredbe grananja), obično je manji od 6 (vrlo mali) pa treba tražiti *ILP* van osnovnog bloka tj. instrukcije iz različitih blokova. Najčešće je dobro da se iskoristi paralelizam koji postoji između različitih iteracija petlje koje su nezavisne jedna od druge.

Primer:

```
FOR i:=1 to 1000 do
X(i)=X(i)+Y(i)
```

Ovo je tipičan primer nezavisne instrukcije. Teoretski bi moglo sa 1000 hardverskih jedinica da se cela petlja izvrši odjedared. Ovaj paralelizam se naziva **L**oop **L**abel **P**aralelism ili **LLP**. I on treba da se transformiše u *ILP*. Svi metodi se za ovo obično svode na **odmotavanje petlje**.

Odmotavanje petlje može da se izvede statički i dinamički. **Statičko odmotavanje** petlje izvodi se kompilatorom a **dinamičko odmotavanje petlje** tokom izvršenja programa.

Da bi protočni sistem bio bez hazarda treba da instrukcije koje su zavisne po podacima budu međusobno razdvojene. Kompajler treba da nađe instrukcije koje nisu međusobno zavisne i da ih ubaci tako da hazard ne uzrokuje zaustavljanje protočnog sistema.

ILP želimo da povećamo da bismo izbegli zastoje zbog hazarda. Da ne bi došlo do zastoja u protočnom sistemu, zavisne instrukcije treba razdvojiti nekim nezavisnim instrukcijama. Ovaj posao treba da odradi kompajler tj. on treba da nađe koje su to naredbe koje će iskoristiti za umetanje i koliko ih treba biti.

Latentnost instrukcija je broj internih klok ciklusa koji treba da prođe između instrukcije koja generiše rezultat i instrukcije koja koristi taj rezultat.

Period instrukcije jednak je broju ciklusa takta koji treba da protekne između izvršavanja dve instrukcijeistog tipa. Ako se ovaj period je ispoštuje može doći do strukturalnog hazarda.

Primer za povećanje ILP–a odmotavanjem petlje (kompilator to radi)

Instrukcija koja generise rezultat	Instrukcija koja koristi rezultat	Latencija
FPALU	FPALU	3
FPALU	SD	2
LD	FPALU	1
LD	SD	0

```
FOR i:=1 to 1000 do
X(i)=X(i)+S
```

R1 sadrži adresu X(1000)

S je u F2

Ako je R1==0 tada se završava petlja.

Dakle,

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

Sa latencijama datim u tabeli, da vidimo kolko je trajanje jedne instrukcije.

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

Dakle, prazna polja predstavljaju zastoje, tako da da jedna instrukcija traje onoliko ciklusa takta koliko ima redova u programu, računajući i zastoje. Dakle, 9 ciklusa takta. Treba videti da li je moguće preurediti instrukcije u cilju smanjenja broja ciklusa takta potrebnih da se izvrši jedna instrukcija.

Preuređen program bi izgledao ovako!

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

lako nam se čini da je lako, ovo nije trivijalno preuređenje i traži moćnije kompilatore.

Dakle, ovo je preuređenje je skratilo potreban broj ciklusa takta na 6.

Ovde je odnos korisnih instrukcija (instrukcije koje vrše neka izračunavanja) i ovih drugih po ciklusu takta jednak 3:3 dakle, treba povećati efikasnost za šta služi odmotavanje petlje. Odmotavanje petlje je umnožavanje petlje i neki delovi se izbacuju (naprimer neka trivijalna izračunavanja NPR: SUBI). Ako se odmotavanje vrši sa istim registrima ne dolazi do velikog povećanja ILP–a. To znači da treba koristiti različite registre.

LOOP:	LD	F0,0(R1)	
	ADDD	F4,F0,F2	
	SD	(R1),F4	
	LD	F6,–8(R1)	
	ADD	F8,F6,F2	
	SD	–8(R1),F8	
	LD	F10,–16(R1)	
	ADD	F12,F10,F2	
	SD	–16(R1),F12	
	LD	F14,–24(R1)	
	ADDD	F16,F14,F2	
	SD	–24(R1),R16	
	SUBI	R1,R1,#32	

	BNEZ	R1,LOOP	

Na primer:

Prikazati prethodnu petlju odmotanu tako da postoje 4 kopije tela petlje. Usvajamo da je R1 umnožak od 32 što znači broj instrukcija je umnožak od 4. Eiminisati bilo koja trivijalna izračunavanja i ne koristiti iste registre više puta.

Vidimo da se ovde koriste 27 ciklusa takta, a da je jedna instrukcija prosečno za 6.8 ciklusa takta.

Naravno ova grdosija od programa je samo odmotana ali ne i preuređena petlja. Sada bi trebalo preurediti petlju

Preuređena petlja bi izgledala ovako:

Dakle, sada imamo 14 ciklusa takta. 14:4 = 3.5

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	–24(R1),F16

Nadam se da se odavde može izvući zaključak da što je veći broj registara to je broj ciklusa takta manji. Tj. biće manje neminovnih zastoja.

Postoji 3 tipa zavisnosti u programu.

1. **Prave zavisnosti**, odgovaraju RAW hazardima
2. **Zavisnosti imenovanja**, koje se pak dele na:
   - Antizavisnosti ili WAR hazarde
   - Izlazne zavisnosti ili WAW hazarde

   Oba ova problema se rešavaju premošćavanjem.
3. **Kontrolne zavisnosti**

Prave zavisnosti:

Između instrukcija I i J gde je I < J postoji zavisnost ako:

- Instrukcija J koristi rezultat instrukcije I
- Instrukcija J zavisi od instrukcije K a instrukcija K od instrukcije I (prave zavisnosti postoje ako postoji lanac zavisnosti prvog tipa)

Ovaj lanac, može biti dug kao i sam program (to nije ništa čudno, čak je i uobičajeno).

Na primer:

LOOP	LD	F0,0(R1)
	ADDD	F4,F0,F2
	SD	0(R1),R4
	SUBI	R1,R1,8
	BNEZ	R1,LOOP

Zavisnosti se javljaju zbog naredbi tj. njihovih operanada koji su označeni istom bojom.

Tj. između (F0–F0, F4–F4, R1–R1)

Zavisnosti po podacima su svojstvo programa, i da li će zavisnost dovesti do hazarda koji se detektuje u protočnom sistemu i da li će to dovesti do zastoja zavisi od konkretne implementacije protočnog sistema. Zavisnosti određuju redosled po kome instrukcije moraju biti izvršavane, ukazuju na mogućnost potencijalnih hazarda i definišu **gornju granicu** ILP–a koji se može eksploatisati.

Između SUBI i BNEZ postoji zavisnost ali to ne dovodi do zastoja ako nam protočni sistem ima premošćavanje. Ove zavisnosti treba poznavati jer one određuju redosled po kome instrukcije moraju biti izvršavane, ukazuju na mogućnosti eventualnog hazarda, definišu gornju granicu paralelizma na nivou instrukcije (ILP) koji se može eksploatisati u programu. Nama je cilj da se ta ograničenja koja unose zavisnosti uklone i to se postiže na dva načina.

1. Izbegavanje hazarda zadržavanjem zavisnosti
2. Eliminisanje zavisnosti transformacijom koda

Ova prva metoda (Izbegavanje hazarda zadržavanjem zavisnosti) se obično izvodi preuređenjem koda i razdvajanjem instrukcija koje su međusobno zavisne, dok se hazard ne izbegne. Što se preuređenja tiče, ono može biti statičko i dinamičko. **Statičko** se izvodi u fazi prevođenja, a **dinamičko** u toku izvršavanja programa.

Dinamičko planiranje izvršenja instrukcija

Do sada smo usvajali da se izdavanje i izvršavanje instrukcija vrši po redosledu pribavljanja. Pribavljanje i izvršavanje se odvija sve dok se ne detektuje **hazard**. Hazard je uglavnom moguće otkloniti premošćavanjem, ali postoje i oni koji se na taj način ne mogu otkloniti, pa se po njihovom otkrivanju protočni sistem mora zaustaviti počev od instrukcije kod koje je detektovan hazard. Sve instrukcije koje slede iza naredbe u kojoj je detektovan hazard biće takođe zaustavljene, sve dok se hazard ne otkloni.

Ako ima dovoljno hardverskih resursa, i ako nema pravih zavisnosti, zaustavljanje se može izbeći. Osnovno ograničenje kod ove metode je da se instrukcije izdaju po redosledu pribavljanja. Ako postoji zavisnost između dve instrukcije koje su bliske, ta zavisnost će izazvati zaustavljanje protočnog sistema.

Primer:

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

Crvenom bojom je označen hazard *RAW* tipa (F0–F0).

Kada se hazard između prve i druge naredbe detektuje, protočni sistem se zaustavlja. Instrukcija SUBD ne zavisi od prethodnih instrukcija i mogla bi da se pusti dalje! Osnovna ideja kod dinamičkog planiranja je da se dozvoli instrukciji da uđe u EX fazu van redosleda. Ovo ima za posledicu i njihovo vanredosledno okončanje zar ne ? Da bi smo ovo obezbedili neophodno je ID fazu podeliti na dve faze. Prva faza je takozvana **ISSUE** faza. U ovoj fazi se vrši dekodiranje i provera strukturalnih hazarda (tj. da li postoji slobodna funkcionalna jedinica koja može prihvatiti tu instrukciju ili ne). Druga faza je **READ OPERANDS** faza. U ovu fazu se instrukcija i ISSUE faze propušta samo ako nije detektovan strukturalni hazard. Sama reč nam dovoljno govori o ovoj fazi! U ovoj fazi se čitaju izvrorišni operandi potrebni za izvršenje instrukcije.

ISSUE faza sledi iza faze pribavljanja instrukcije (IF), pri čemu se pribavljene instrukcije smeštaju u instrukcioni bafer. Iza toga sledi ISSUE pa READ OPERANDS faza, pa EXE, (nećmo razmatrati MEM fazu jer govorimo o instrukcijama koje rade sa podacima u pokretnom zarezu kojise nalaze u F#registrima) i na kraju ide WB faza.

SCOREBOARD

Prevod ove engleske reči na Srpski jezik bi bio **Evidenciona tabela**. Mada je prevod veoma dobar, mi ćemo ipak koristiti termin **Scoreboard**. Scoreboarde tehnika je tehnika koja koristi dinamičko planiranje izvršenja instrukcija. Od fundamentalnih novina kod Scoreboard–a srećemo se sa pojmom **neredosledno izvršavanje**, koje za sobom povlači i podelu ID stepena (faze) na dve nove podfaze. Te faze su ISSUE i READ OPERANDS. U uvodnom delu o ovoj podeli smo razgovarali i istakli osnovne ideje.

Prvi put je primenjena 1963 kod CDC 6600 koji je imao 16 funkcionalnih jedinica (4 za FP, 5 za MEM, 7 za INT podatke). Sve ovo podrazumeva da više instrukcija sme da bude istovremeno u EXE fazi (upravo zato imamo više funkcionalnih jedinica). Dakle, instrukcije se izvršavaju kad god ne zavise od nekih drugih (predhodnih) i naravno, ako uz to ne postoji hazard.

Dakle, kod CDC 6600 računara po prvi put se javljaju pojmovi:

- Neredosledno izdavanje
- Neredosledno izvršenje
- Neredosledno kompletiranje

Po prvi put se kod ovog računara NE projektuje hardver za premošćavanje, zato što premošćavanje nije potrebno.

U **ISSUE** fazi se vrši dekoditranje instrukcija i provera postojanje strukturalnih hazarda. Instrukcije prolaze po redu pribavljanja kroz ISSUE fazu (**in order issue**). Ovo redosledno izdavanje instrukcija je obavezno, jer se tako može lako otkriti hazard. Odnosno, **ukoliko postoji strukturalni hazard instrukcija se ne izdaje**. Jasno je da se još u ovoj fazi otklanja opasnost od strukturalnih hazarda. U ovom stepenu instrukcija može biti zaustavljena ili propuštena (ako nema hazarda) u izvršenje van redosleda (**OUT OF ORDER EXECUTION**).

Ona se zaustavlja ako joj operandi nisu dostupni (*detektovan je RAW hazard*), naravno, ako su joj operandi dostupni, ona se pušta u izvršavanje. Dakle, izvršavanje počinje, čim joj operandi postanu dostupni. Međutim, na ovaj način se mogu pojaviti i **antizavisnosti i izlazne zavisnosti**. O ovim zavisnostima brine Scoreboard.

Dakle, rešenja primenjena u Scoreboard tehnici dovela su do toga da se usled neredoslednog kompletiranja omogući veća šansa za nastajnje **WAR** i **WAW** hazarda. Eto nama novih problema! Međutim, **WAR** hazardi se mogu otkloniti tako što se zaustavi upis u registar do očitavanja registra ili tako što će se registri čitati samo u stepenu READ OPERANDS. Što se tiče WAW hazarda njihovo rešenje se svodi na detektovanje hazarda i zaustavljanje izdavanja nove instrukcije do kompletiranja prethodne (tekuće) instrukcije. Važno je pomenuti da kod Scoreboard tehnike nema preimenovanja registara.

Kada bi hteli da o Scoreboardu kažemo ono najbitnije, to bi se svelo skoro na sam prevod ove reči, a to je vođenje evidencije o zavisnostima između instrukcija koje su već izdate. Scoreboard zamenjuje stepene ID,EX i WB sa četiri nova stepena (zato što je ID fazu podelio na dve faze, sećate se uvodnog dela o dinamičkom planiranju izvršenja instrukcija?).

Primer:

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

SUBD	F10,F8,F14

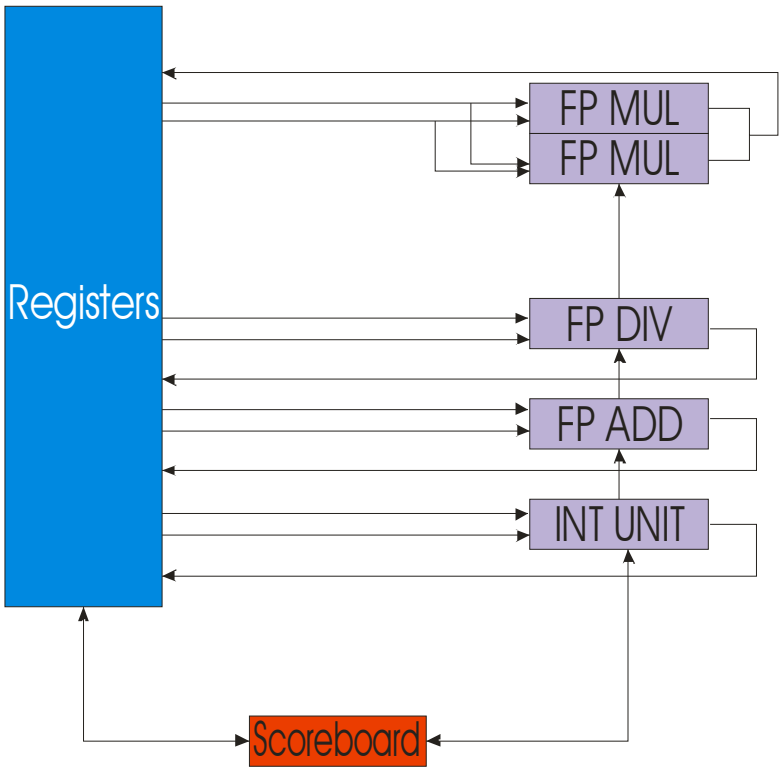
Primetimo da je poslednja, treća, linija programa napisane u dve varijante. Tj. ukoliko bi treća linija bila kao prva varijanta, onda bi nastali WAR hazardi, a ukoliko bi bila kao drugoponuđena varijanta onda bi nastao WAW hazard.

Ako se dozvoli da SUBD uđe u izvršenje i upiše rezultat to znači da će izvršenje programa biti nekorektno. Naš zadatak je da otkrijemo te zavisnosti!

U idealnom slučaju SCOREBOARD tehnika treba da obezbedi 1 instrukciju u jednom ciklusu takta kada nema strukturalnih hazarda i hazarda popodacima.

Primer:

(Procedura sa 5 funkcionalnih jedinica! I to 2 FP množača, 1 FP delitelj, 1 FP sabirač i 1 INT.



Svaka instrukcija koja se pribavi prolazi kroz **SCOREBOARD** gde se pravi zapis o svim zavisnostima po podacima i strukturalnim zavisnostima. Scoreboard određuje da li i kada instrukcija može pribaviti operande i nastaviti sa izvršenjem, i kada instrukcija može da upiše rezultat u odredišni egistar. Na taj način je detekovanje hazarda i upravljanje **lokalizovano** u Scorebordu–u.

Umesto IF, ID, EX, MEM i WB faza, sada imamo sledeće faze.

IF

Pribavljanje instrukcije (normalno)

ISSUE

U ovoj fazi se proveravaju strukturalni hazardi tj. da li je željena funkcionalna jedinica slobodna, i utvrđuje se izlazna zavisnost tako što se proverava da li postoji neka druga aktivna instrukcija koja ima isti odredišni registar. **Ako postoji WAW hazard ili ako funkcionalna jedinica nije slobodna, instrukcija se ne propušta u sledeći stepen**. To uzrokuje upis u **bafer** između IF i ISSUE stepena. Kada se ovaj bafer napuni prestaće dalje pribavljanje novih instrukcija. Kada nema WAW hazarda i kada je funkcionalna jedinica slobodna, instrukcija se propušta u skledeći stepen.

READ OPERANDS

U ovoj fazi se ispituje raspoloživost izvornih operanada. Operand je **raspoloživ** ako u sistemu ne postoji ni jedna aktivna instrukcija koja će ga generisati. U ovom stepenu se RAW hazardi rešavaju dinamički tako što se instrukcija može propustiti u izvršenje van redosleda ako su joj operandi dostupni. Ako pak postoji RAW hazard (makar jedan od operanada nije dostupan) instrukcija se zaustavlja.

EXECUTION

U ovoj fazi se obavlja izvršavanje instrukcije. Scoreboard je obavestio funkcionlnu jedinicu kada može da pročita operande. Nakon završetka operacije, funkcionalna jedinica obaveštava Scoreboard da se završila.

WRITE RESULT

U ovoj fazi Scoreboard proverava da li postoji antizavisnost (**WAR** hazard). Ako se ona detektuje zabranjuje se upis u registar dok se hazard ne otkloni. Pošto se operandi čitaju samo kada su oba dostupna Scoreboard ne koristi premošćavanje. Ovo nije veliko ograničenje jer se instrukciji dozvoljava da izvrši upis u registarsko polje čim se ona završi (ako nema WAR hazarda), pa premošćavanje ne bi donelo poboljšanje.

Instruction status table:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F6,34(R2)				
LD F2,45(R3)				
MULD F0,F2,F4				
SUBD F8,F6,F2				
DIVD F10,F0,F6				
ADD F6,F8,F2				

U ovoj tabeli se za svaku instrukciju koja je u Scoreboardu vodi evidencija o tome u kojoj se fazi nalazi (izvršene faze su označene ljubičastom bojom). Tu može biti onoliko instrukcija koliko ima mesta u Scoreboard–u.

Ako gledamo prvu LD instrukciju, vidimo da su se sve njene faze izvršile. Znači, ona se korektno izvršila, i upisala je rezultat u registar. Ako međutim posmatramo drugu LD instrukciju, ona se takođe izvršila, ali nije upisala svoj rezultat u registra. Što se tiče instrukcija MULD, SUBD, DIVD, one su sve izdate, ali su još ostale u fazi ISSUE. Dakle, zakočene, čekajući svoje operande.

NPR: Instrukcija ADD nije prošla ni issue fazu jer postoji samo jedan FP sabirač, a u njemu se već nalazi SUBD. Nadam se da je jasno :–)

Functional unit status table:

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Div	Yes	Div	F10	F0	F6	Mult1		No	Yes

Za svaku funkcionalnu jedinicu vodi se evidencijai. Ova tabela se sastoji od 9 polja:

- Busy — Označava da li je funkcionalna jedinica slobodna ili ne.
- Op — Pamti operaciju koju funkcionalna jedinica trenutno obavlja.
- Fi — Je polje odredišnog registra (oznaka registra)
- Fj i Fk — Su polja izvorišnih registara
- Qj i Qk — Kazuju koja funkcionalna jedinica treba da generiše rezultat koji će biti upisan u F_j odnosno F_k (ako postoji takva funkcionalna jedinica, a ako ne postoji operandi su već dostupni u registrima).
- Rj i Rk — Govore o tome kada su izvorišni operandi Fj i Fk dostupni. Kada se prođe kroz READ OPERANDS fazu oba polja se brišu tj. postavljaju na (No) pošto su posle toga nebitna (do tada je no što znači da operandi nisu dostupni).

Ova tabela nam govori da: prva naredba množenja mult1 čeka na rezultat iz intger funkcionalne jedinice. Takođe ADD jedinica čeka na rezultat iz integer jedinice, a DIV jedinica čeka na rezultat iz prve mult jedinice mult1. ADDD instrukcija je zakočena zbog strukturalnog hazarda. Ovaj hazard će biti otklonjen kada SUBD završi svoje izvršavanje.

Primećujemo da su neka mesta u ovim tabelama ostavljena orazna. To znači da su te funkcionalne jedinice neupotrebljene (u tom trenutku). NPR: Rk polje nije popunjeno za LOAD instrukciju, a ovo polje je takođe ostalo prazno za MULT2 jedinicu, zbog toga što ovo polje nema značenje za te naredbe. Takođe, kada su operandi pročitani Rj i Rk polja su setovana na NO.

Register result status table:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1	integer			Add	Div					

Za svaki od registara opšte namene stoji oznaka funkcionalne jedinice koja trenutno radi obradu i izvršiće upis u taj registar (ako nema takve funkcionalne jedinice onda je polje za taj registar prazno).

Integer unit služi za ALU operacije, grananja, load i store.

Rezime:

Ove tri tabele su detaljna struktura podataka koja predstavlja evidenciju koju vodi Scoreboard pri izvršenju svake instrukcije.

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

Plavom i crvenom bojom su prikazane prave zavisnosti, a zelenom bojom antizavisnosti.

Ovo je ilustrovan primer kako nastaju hazardi. Ovaj primer poslužiće nam za shvatanje principa rada Scoreboard–a.

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F6,34(R2)				
LD F2,45(R3)				
MULD F0,F2,F4				
SUBD F8,F6,F2				
DIVD F10,F0,F6				
ADDD F6,F8,F2				

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	Yes	Div	F10	F0	F6			No	No

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

FU						Div					

Ove tri tabele prikazuju Scoreboard, u trenutku kada DIVD, samo što nije upisao rezultat. ADDD je sada u mogućnosti da okonča izvršavanje zato što je DIVD prošla kroz READ OPERANDS fazu, tj. sada je ADDD dobila kopiju F_6 registra. Dakle, ostaje samo da se DIVD završi.

Simulacija Scoreboard–a

Sada ćemo pokušati da simuliramo ScoreBoard tehniku!

Pre toga moramo usvojiti sledeće latencije (ADD=2, MUL=10, DIV=40)

Ovu simulaciju počećemo polazeći od stanja koje smo opisali sa sledeće tri tabele.

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)				
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADD F_6,F_8,F_2				

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Div	Yes	Div	F10	F0	F6	Mult1		No	Yes

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1	integer			Add	Div					

Dakle, počinjemo simulaciju od trenutka kada su MULD i DIVD spremne da uđu u Write Result fazu.

Javljaju se RAW hazardi po podacima između:

- Druge LD naredbe i MULD, ADDD, SUBD
- MULD i DIVD
- SUBD i ADD

Javljaju se WAR hazardi između:

- DIVD i ADDD, SUBD

Javlja se takođe i strukturalni hazard na ADD funkcionalnoj jedinici za ADD i SUBD instrukcije.

Imajući sve ovo u vidu, prikazaćemo kako će tabele izgledati kada MULD i DIVD budu spremne da upišu rezultat u registre na sledećim tabelama (respektivno).

Instruction ststus table:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)				
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Div	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1			Add		Div					

Ovo su tabele koje prikazuju trenutak kada se MULD sprema da upiše rezultat u registar.

DIVD još nije ni pročitala svoje operande zato što oni zavise od rezultata množenja. Što se tiče ADD instrukcije, ona je pročitala svoje operande i trenutno je u fazi izvršenja mada je bila prinuđena da čeka dok SUBD ne završi i oslobodi funkcionalnu jedinicu. ADDD neće moći da upiše rezultat zbog postojanja WAR hazarda na registru F_6, koga koristi DIVD.

Q polja su važna (imaju smisla) samo onda kada neka funkcionalna jedinica čeka na neku drugu funkcionlnu jedinicu.

Sledeće tri tabele prikazuju situaciju kada je DIVD spremna da upiše svoj rezultat.

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)				
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	Yes	Div	F10	F0	F6			No	No

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU						Div					

Ove tri tabele prikazuju Scoreboard, u trenutku kada DIVD, samo što nije upisao rezultat. ADDD je sada u mogućnosti da okonča izvršavanje zato što je DIVD prošla kroz READ OPERANDS fazu, tj. sada je ADDD dobila kopiju F_6 registra. Dakle, ostaje samo da se DIVD završi.

Sada možemo da vidimo kako Scoreboard radi do detalja, ako pogledamo šta treba da se desi da bi Scoreboard pustio instukciju da se dalje izvrši. To je prikazano sledećim algoritmom.

Instruction Status	Wait Until	Bookkeeping
ISSUE	Not Busy (FU) and not Result (D)	Busy(FU)<–yes; Op(FU)<–op; F_i(FU)<–0; F_j(FU)>–S1; F_k(FU)<–S2; Q_j<–result(S1); Q_k<–result(S2); R_j<–Not Q_j; R_k<–Not Q_k; result(D)<–FU;
READ OPERANDS	R_j and R_k	R_j<–No; R_k<–No; Q_j<–0; Q_k<–0
EXECUTION COMPLETE	Functional unit done	
WRITE RESULT	Za svako f(($F(f)<>F_i(FU)$ or $R_j(f)$ = No & ($F_k(f)<>F_i(FU)$ or $R_k(f)$=No))	Za svako f(if $Q_j(f)$=FU then $R_j(f)$<–yes; Za svako f(if $Q_k(f)$=FU then $R_k(f)$<–yes; Result(F_i(FU)) <– 0; Busy(FU) <– No

FU je funkcionalna jedinica koju koristi tekuća instrukcija. D je ime odredišnog registra, S1 i S2 su imena izvrorišnih registara, a op je operacija koja se izvršava. Što se notacije tiče objasnićemo je na sledećem primeru. NPR: Ukoliko želimo da pristupimo F_i–u iz Scoreboard–a za funkcionalnu jedinicu FU, to ćemo uraditi na sledeći način. F_i(FU). Result(D) je ime funkcionalne jedinice koja će upusati u registar D. Testiranje u WRITE RESULT fazi sprečava upis ukoliko postoji WAR hazard. WAR hazard, da podsetimo, postoji ukoliko neka druga instrukcija koristi koristi destinacioni registar tekuće naredbe (F_i(FU)) kao svoj izvorišni (F_j(f)). taođe, ovaj test sprečava upis ukoliko je neka druga instrukcija upisala registar (R_j=Yes ili R_k=Yes). Uvde je sa f označena promenljiva koja predstavlja bilo koju funkcionalnu jedinicu.

Za kraj rasprave o Scoreboard tehnici, proćićemo kroz jedan primer i tu r4azjasniti sve što nije bilo jasno u toku rasprave. Dakle, još jedared ćemo simulirati rad Scoreboard–a.

Usvojićemo da su latencije funkcionalnih jedinica:

- Integer = 1
- Add = 2
- Mult = 10
- Div = 40

Na početku su sve tabele Scoreboard–a prazne:

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)				
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

FU											

Sada krećemo u prvi ciklus (Clock=1)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1			
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Izdaje se prva LOAD naredba.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	LOAD	F_6		R_2				Yes
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Integer funkcionalna jedinica postaje zauzeta (busy), i to izvršavajući LOAD instrukciju.

Destinacioni registar je F_6, a kao izvorišni (S_2) se koristi R_2.

Polja R_j i R_k govore o tome da li su operandi F_j i F_k dostupni! A oni su u našem slučaju dostupni.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU				Integer							

Ovaj podatak nam kazuje da će Integer funkcionalna jedinica upisati u registar F_6.

Sada krećemo u ciklus (Clock=2)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2		
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Druga LOAD naredba ne može da se izda zato što se prva još nije izvršila (latencija load inmstrukcije je 1). A sistem ima samojednu INTEGER funkcionalnu jedinicu.

Dakle, pošto su prvoj naredbi operandi bili dostupni (u prošlom trenutku) ona sada pribavlja svoje operande.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	LOAD	F_6		R_2				Yes
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Ova tabela ostaje ista, zato što je još uvek u toku izvršavanje prve instrukcije

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU				Integer							

Naravno i ova tabela ostaje ista.

Sada krećemo u ciklus (Clock=3)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

U trećem ciklusu će se završiti prva LOAD instrukcija. Tj. ućiće u fazu EXECUTION COMPLETE zato što je njena latencija (latencija INTEGER funkcionalne jedinice upravo 1 clk).

Druga LOAD naredba nije mogla da se izda zato što je INT funkcionalna jedinica bila zauzeta.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	LOAD	F_6		R_2				No
Mult1	No								
Mult2	No								
Add	No								
Div	No								

I dalje je zauzeta Integr funkcionalna jedinica, i dalje je destinacioni registar F_6 a drugi izvorišni (S2) je R_2.

Pošto se u ovoj fazi čita registar R_2, on postaje nedostupan za ostale instrukcije. Dakle R_k=no

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU				Integer							

Ova tabela nam govori o tome, kako i dalje F_6 očekuje da bude upisan od strane integer funkcionalne jedinice.

Sada krećemo u ciklus (Clock=4)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)				
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

U četvrtom ciklusu takta prva LOAD instrukcija upisuje svoj rezultat i time potpuno završava svoj životni ciklus.

Dakle, sada je oslobodila INT funkcionalnu jedinicu.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Ova tabela je prazna zato što nijedna Funkcionalna jedinica ne izvršava ni jednu operaciju.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU											

Pošto u ovom trenutku prva load instrukcija upisuje u svoj odredišni registar, a nijedna druga nije ni pribavljena, jasno je zašto nijedan registar ne očekuje da bude upisan.

Sada krećemo u ciklus (Clock=5)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5			
MULD F_0,F_2,F_4				
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Sada se oslobodila integer funkcionalna jedinica i druga LOAD instrukcija može da krene u ISSUE fazu tj. da se izda.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	LOAD	F_2		R_3				Yes
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Naravno, integer funkcionalna jedinica postaje ponovo zauzeta vršeći operaciju LOAD. Destinacioni registar je sada F_2, a drugi izvorišni (load naredbe imaju samo drugi izvorišni) je R_3. R_k nam kazuje da su svi izvorišno operandi dostupni.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU		Integer									

Sada registar F_2 očekuje upis iz integer funkcionalne jedinice.

Sada krećemo u ciklus (Clock=6)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6		
MULD F_0,F_2,F_4	6			
SUBD F_8,F_6,F_2				
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

Druga LOAD instrukcija, može da krene u fazu čitanja operanada jer su joj operandi (operand) bili dostupni. Međutim, u isto vreme i naredba MULD može da krene u izdavanje jer ne koristi zajedničke hardverske resurse.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	LOAD	F_2		R_3				Yes
Mult1	Yes	MULT	F_0	F_2	F_4	Integer		No	Yes
Mult2	No								
Add	No								
Div	No								

Dakle, sada je izdata i MULT naredba. Ona kao destinacioni registar ima F_0 a izvorišni registri su F_2 i F_4. Što se tih registara tiče, F_4 je dostupan, ali F_2 nije jer ga treba generisati integer funkcionalna jedinica – (Q_j) polje.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult	Integer									

Sada registar F_2 očekuje upis iz integer funkcionalne jedinice a F_0 iz mult funkcionalne jedinice.

Sada krećemo u ciklus (Clock=7)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	
MULD F_0,F_2,F_4	6			
SUBD F_8,F_6,F_2	7			
DIVD F_{10},F_0,F_6				
ADDD F_6,F_8,F_2				

U sedmom ciklusu takta druga LOAD instrukcija ulazi u fazu izvršenja.

Mult ne može da krene u fazu pribavljanja operanada zato što joj operandi nisu bili dostupni. Ali se zato izdaje SUBD naredba, jer će koristiti funkcionalnu jedinicu ADD koja je slobodna.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4	Integer		No	Yes
Mult2	No								
Add	Yes	SUBD	F_8	F_6	F_2		Integer	Yes	No
Div	No								

Druga load instrukcija je završila svoje izračunavanje, pa je oslobodila integer funkcionalnu jedinicu.

Mult naredba i dalje radi (istao kao u prošlom ciklusu).

Add funkcionalna jedinica izvršava SUBD naredbu, ali ni njoj operandi nisu dostupni (čeka na F_2).

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult	Integer			Add						

Sada registar F_2 očekuje upis iz integer funkcionalne jedinice a F_0 iz mult funkcionalne jedinice.

Sada krećemo u ciklus (Clock=8)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6			
SUBD F_8,F_6,F_2	7			
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2				

U osmom ciklusu takta druga LOAD instrukcija upisuje rezultat.

Mult i Subd naredbe ostaju zakočene jer čekaju na F_2. Ali se zato izdaje naredba DIV jer će ona koristiti DIV funkcionalnu jedinicu).

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			Yes	Yes
Mult2	No								
Add	Yes	SUBD	F_8	F_6	F_2			Yes	Yes
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

Druga LOAD instrukcija je upisala rezultat.

MULD sada (pošto je load završila) ima slobodne operande.

Add takođe ima slobodne operande.

Ali Divd čeka na MULT da upiše rezultat, tako da ona još uvek nema slobodne opernade.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult				Add	Div					

F_0 očekuje da bude opisan sa rezultatom MULT naredbe, F_8 sa rezultatom SUBD naredbe, a F_{10} sa rezultatom DIV naredbe.

Sada krećemo u ciklus (Clock=9)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9		
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2				

U ovom ciklusu napreduju samo MULD i SUBD naredbe jer su se njihovioperandi oslobodili tj. postali dostupni. I one čitaju svoje operande.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			Yes	Yes
Mult2	No								
Add	Yes	SUBD	F_8	F_6	F_2			Yes	Yes
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1				Add	Div					

F_0 očekuje da bude opisan sa rezultatom MULT naredbe, F_8 sa rezultatom SUBD naredbe, a F_{10} sa rezultatom DIV naredbe.

Sada krećemo u ciklus (Clock=10)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9		
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2				

U ovom ciklusu multd ne može da završi, jer je njegova latencija 10 kloka.

Takođe nemože ni SUBD ne može da završi jer ona ima latenciju od 2 kloka.

DIVD ne može da pribavlja svoje operande zato što F_0 treba da bude upisano od strane MULTD, a ona nije završila.

ADD ne može sa se izda jer bi nastao konflikt sa naredbom SUBD koja koristi ADD funkcionalnu jedinicu.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			Yes	Yes
Mult2	No								
Add	Yes	SUBD	F_8	F_6	F_2			Yes	Yes
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

U ovok ciklusu nema nikakvih izmena.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1				Add	Div					

Sada krećemo u ciklus (Clock=11)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9	11	
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2				

MULTD ne može da napreduje (latencija=10).

SUBD, nedutim završava jer je njena latencija 2, a pribavila je operande u 9–om ciklusu takta.

DIVD ostaje blokirana jer očekuje rezultat MULD–a.

ADDD i dalje nemože da se izda jer bi nastao konflikt na ADD funkcionalnoj jedinici.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			Yes	Yes
Mult2	No								
Add	No								
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

Pošto je SUBD završila, onda ADD jedinica postaje slobodna.

Ostalo ostaje isto.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1				Add	Div					

Sada krećemo u ciklus (Clock=12)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2				

MULTD ne može da napreduje (latencija=10).

SUBD upisuje rezultat.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			No	No
Mult2	No								
Add	No								
Div	Yes	DIVD	F_{10}	F_0	F_6			No	Yes

Subd upisuje rezultat, to znači da je oslobodila ADD funkcionalnu jedinicu.

MULTD i dalje radi sa operandima, i zato su nedostupni.

DIVD i dalje čeka na MULTD i njegov rezultat, da bi pribavio operande.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1					Div					

Pošto je SUBD upisala rezultat, sada registar F_8 je očekuje upis.

Sada krećemo u ciklus (Clock=13)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2	13			

MULTD ne može da napreduje (latencija=10).

DIVD još čeka na MULD.

ADDD se izdaje, zato što je ADD funkcionalna jedinica slobodna.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			No	No
Mult2	No								
Add	Yes	ADDD	F_6	F_8	F_2			Yes	Yes
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

ADDD zauzima ADD funkcionalnu jedinicu.

MULTD i dalje radi sa operandima, i zato su nedostupni.

DIVD i dalje čeka na MULTD i njegov rezultat, da bi pribavio operande.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1			ADD		Div					

Sada registar F_6 očekuje upis.

Preskočićemo neke cikluse jer se nadam da je osnovni princip objašnjen. Sada ćemo skočiti na ciklus takta u kome nastaje hazard. Da bi se upoznali sa takvim situacijama.

Sada krećemo u ciklus (Clock=17)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9		
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2	13	14	16	

Muld još nije završila.

Divd čeka na Muld.

ADDD je završila i može da upiše rezultat ALI ???

DIVD–u treba F_6 onakav kakav je sada, a ne onakav kakvog će ga promeniti naredba ADDD zato što je naredba ADDD izvršena neredosledno. Dakle, sistem se koči da bi se otklonio hazard.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			No	No
Mult2	No								
Add	Yes	ADDD	F_6	F_8	F_2			No	No
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1			ADD		Div					

Sada krećemo u ciklus (Clock=19)

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9	19	
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8			
ADDD F_6,F_8,F_2	13	14	16	

Multd se upravo završila (nakon 10 ciklusa kloka).

Divd i dalje čeka da MULD upiše rezultat.

Add i dalje nesme da upiše rezultat jer je još uvek aktuelan hazard. Hazard će nestati kada DIVD pribavi svoje operande.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	Yes	MULT	F_0	F_2	F_4			No	No
Mult2	No								
Add	Yes	ADDD	F_6	F_8	F_2			No	No
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	Yes

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU	Mult1			ADD		Div					

Skočićemo sada u ciklus 21 da bi videli trenutak kada nestaje hazard.

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9	19	20
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8	21		
ADDD F_6,F_8,F_2	13	14	16	

MULD je još u prošlom (20–tom) ciklusu upisala rezultat. Znači, sada divd može da pribavlja svoje operande.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	Yes	ADDD	F_6	F_8	F_2			No	No
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		Yes	Yes

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU				ADD		Div					

Idemo sada u ciklus 22

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9	19	20
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8	21		
ADDD F_6,F_8,F_2	13	14	16	22

Pošto se hazard otklonio, ADDD sada upisuje rezultat.

Ostaje samo da DIVD sačeka svoju latenciju i prođe kroz sledeće faze.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	Yes	DIVD	F_{10}	F_0	F_6	Mult1		No	No

Dok se operandi čitaju, oni su nedostupni ostalim naredbama (kojih ovde nema, ali nema veze ipak su nedostupni).

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU						Div					

Skačemo sada u ciklus 62 gde se ceo program okončava. Da bismo videli i tu situaciju.

Instruction status:

Instruction	ISSUE	READ OPERANDS	EXECUTION COMPLETE	WRITE RESULT
LD F_6,34(R_2)	1	2	3	4
LD F_2,45(R_3)	5	6	7	8
MULD F_0,F_2,F_4	6	9	19	20
SUBD F_8,F_6,F_2	7	9	11	12
DIVD F_{10},F_0,F_6	8	21	61	62
ADDD F_6,F_8,F_2	13	14	16	22

Sve instrukcije su se završile.

Function unit status:

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Div	No								

Tabela je ponovo prazna kao na početku.

Register result status:

	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
FU											

Ovim detaljnim primerom sam nadam se pokazao kako radi Scoreboard.

Scoreboard tehnikom se postiže ubrzanje ok oko 1.7 puta za programe koje kompilator preuređuje, a čak i do 2.5 puta za ručno kodirane programe.

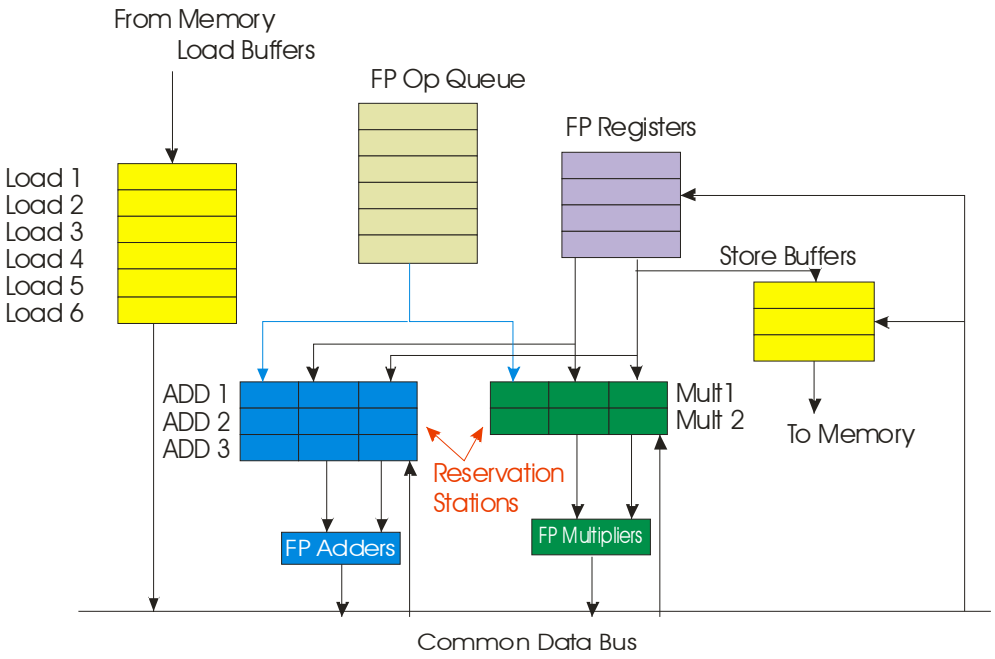
Loše strane Scoreboard–a su:

- Spora memorija (nma keša)
- Nema premošćavanja
- Ograničenje na instrukcije u osnovnom bloku (mali prozor)
- Mali broj funkcionalnih jedinica (pa se javljaju strukturalni hazardi)
- Nema izdavanja pri strukturalnim hazardima
- Čekanje pri WAR hazardima

Tomasulov algoritam

Dobio je ime po Robertu Tomasulu.

To je tehnika koja je prvi put primenjena kod IBM 360/91. Tri godine nakon Scoreboard–a! Svrha njegovog nastanka je bila da postigne visoke performanse bez suviše komplikovanih kompilatora, kao što je to bio slučaj sa Scoreboardom. Koristi ga Pentium II računar. Omogućava da **instrukcije uđu u EXE fazu van redosleda**. Kombinuje ključne elemente Scoreboarda i tehniku preimenovanja registara radi eliminisanja WAR i WAW hazarda. Jedna varijanta za eliminisanje WAR i WAW hazarda je da preimenovanje izvrši kompilator, ali za to je potrebno imati veliki broj registara opšte namene. IMB 360/51 ima samo 4 registra opšte namene pa ovo nije bilo moguće. Staza podataka na kojoj je implementiran Tomasulov algoritam data je na slici.



FP instrukcije koje se pribavljaju iz memorije smeštaju se u **red čekanja** (FP op Queue), odakle se vrši izdavanje instrukcija. Postoje posebne funkcionalne jedinice za FPLOAD i FPSTORE (one na slici nisu označene ali su označeni FPBaferi). Do 6 LOAD i 3 STORE instrukcije mogu bit istovremeno aktivne. Registarski fajl ima samo 4 registra opšte namene. Postoji 5 funkcionalnih jedinica za operacije u pokretnom zarezu (pored onih za LOAD i STORE), 3 za FP sabiranje i oduzimanje i 2 za FP množenje i deljenje. Svakoj funkcionalnoj jedinici pridružena je po jedna rezervaciona stanica. Ona upravlja radom funkcionalne jedinice i izvršenjem instrukcija. Rezervaciona stanica pamti instrukciju koja je izdata (pribavljena iz reda za čekanje) i operande te instrukcije, ako su dostupni (ne imena registara, već same vrednosti). Ako operandi nisu dostupni ovde se pamte oznake funkcionalnih jedinica koje će generisati operande, kao i još neke relevantne informacije za upravljanje izvršenjem instrukcije.

FP registri su parom magistrala povezani sa rezervacionim stanicama i jednom magistralom sa STORE baferima. Svi FP registri, sve rezervacione stanice i svi STORE i LOAD baferi su povezani sa **CDB**–om (**C**ommon **D**ata **B**us). Na *CDB* se šalju rezultati LOAD instrrukcije i rezultati funkcionalnih jedinica. Čim je rezultat dostupan na CDB–u, bilo koja funkcionalna jedinica kojoj je taj podatak potreban može da ga pročita sa CDB–a (ne mora da čeka upis u registarski fajl, kao što je to bio slučaj kod Scoreboard–a).

Detekcija hazarda i upravljanje izvršenjem instrukcija je distribuiraneo po rezervacionim stanicama (kod Scoreboard–a je upravljanje i detekija hazarda bilo cenralizovano.

Otklanjanje WAW i WAR hazarda se vrši preimenovanjem registara.

Kod WAW hazarda (izlazna zavisnost) i WAR hazarda (Antizavisnost) nema toka podataka od prve instrukcije para, ka drugoj instrukciji para. Zato se za ove instrukcije kaže da nisu prave zavisnosti. Ove zavisnosti se mogu otkloniti ako se u drugoj instrukciji odredišni registar zameni nekim drugim registrom. Crvenom bojom je su obeleženi registri koji će se preimenovati zbog WAW hazarda.

Primer

ADD	R4,R5,R6	ADD	R4,R5,R6
LD	R4,56(R8)	LD	R10,56(R8)
SUB	R8,R5,R9	SUB	R11,R5,R9

Faze kod Tomasulovog algoritma su:

Na početku, se instrukcija nalazi u IF fazi, u kojoj se instrukcija ubacuje u red čekanja (FIFO struktura). Posle ove faze slede 3 faze.

1. **ISSUE** instrukcija se pribavlja iz reda čekanja i ako je u pitanju FP operacija ona se izdaje ako postoji slobodna rezervaciona stanica a ako je u pitanju LOAD/STORE izdaje se ako postoji slobodan LOAD/STORE buffer. **Ako ne postoji slobodna rezervaciona stanica u pitanju je strukturalni hazard i izdavanje instrukcije se zabranjuje**. U ovoj fazi se eliminišu antizavisnosti (WAR) i izlazne zavisnosti (WAW) preimenovanjem registara (umesto imena registara koristimo imena odgovarajućih funkcionalnih jedinica).
2. **EXECUTION**. Ako neki operand nije dostupan, nadgleda se CDB. **Kada postane dostupan smešta se u odgovarajuću rezervacionu stanicu**. Kada oba operanda postanu dostupna otpočinje izvršenje FP operacije. Čekanjem da operandi budu dostupni rešavaju se **RAW hazardi**.
3. **WRITE RESULT**. Kad LOAD okonča sa izvršavanjem rezultat se šalje na CDB, rezultat se takođe upisuje svuda gde je potrebno. Rezervaciona stanica se tada oslobađa.

Bitne razlike između Tomasulovog algoritma i Scoreboard–a:

1. Kod *Tomasulovog algoritma* nema provere antizavisnosti i izlaznih zavisnosti jer se ti hazardi eliminišu u ISSUE fazi preimenovanjem registara.
2. CDB se koristi da **emituje** (broadcast–uje) rezultat tako da on odmah postane dostupan svim FJ kojima je potreban, a ne da se prvo vrši upis u RF.
3. LOAD i STORE se tretuiraju kao posebne FJ (IBM–ovi računari su imali veliku latentnost pristupu memoriji, a izdat je pre postojanja keš memorije)

Ključna razlika je u tome što su upravljanje i baferi distribuirani uz funkcionalne jedinice naspram centralizovanih u Scoreboard tehnici. Baferi funkcionalnih jedinica su nazvani rezervacione stanice (čuvaju nerazrešene operande). Registri u instrukcijama zamenjeni su vrednostima ili pokazivačima na rezervacione stanice. Ovo se zove preimenovanje registara. Tomasulov algoritam uspešno izbegava WAR i WAW hazarde. Rezervaciona stanica je više od registra. To omogućava optimizacije koje kompilator (kod scoreboard–a) ne pruža.

Svi podaci potrebni da se detektuju i eliminišu hazardi pridruženi su rezervacionim stanicama, FP registrima i STORE/LOAD baferima. Sve rezervacione stanice, FP registri i LOAD/STORE baferi sadrže takozvani **TAG FIELD** (polje sa oznakom). Kod registara i bafera toje po jedno polje, a kod rezervacionih stanica to je 2 polja. Ono nosi oznaku Funkcionalne jedinice ili LOAD/STORE bafera koji će generisati rezulta (traženi operand). Oznaka 0 znači da je operand već dostupan (ne treba da se čeka). Znači, ili se već nalazi u registarskom fajlu ili je neposredan operand u instrukciji ili se pak uopšte ne koristi.

U ovom primeru dovoljno je za to koristiti polje od 4b. 5FU i 6 Load buffera što je 11, a 2^4 je 16 to znači imamo malo viška, ali mora da bude stepen dvojke. Dakle, imamo 11 virtuelnih registara koji se koriste u fazi preimenovanja registara (u cilju razrešenja WAR i WAW hazarda).

Svaka Rezervaciona stanica koja je pridružena FP funkcionalnim jedinicama, koje obavljaju FP operacije, sadrži 6 polja preko kojih se vodi evidencija o napredovanju instrukcije:

OP — označava koju operaciju obavlja FJ

Qj,Qk — tagovi kod RS–ova označavaju FJ ili LOAD/STORE bufer koji će da generišu rezultat, koji je u stvari operand koji nam treba.

Uj,Uk — Sadržeoperande (baš vrednosti, a ne imena). Ne može da stoji nešto istovremeno i u Uj i Qj zato što je podatak ili dostupan ili se nanjega čeka, trećeg nema.

BUSY — Da li je rezervaciona stanica slobodna ili ne.

Kod registarskog fajla i kod STORE/LOAD buffera postoji jedno tag polje označeno sa Qi koje sadrži oznaku virtuelnog registra i podatke o tome koja Funkcionalna jedinica generiše rezultat za odredišni registar opšte namene.

LOAD/STORE baferi imaju i indikator koji pokazuje da li je bafer pun ili prazan. U ovim baferima se pamti efektivna adresa sa koje se vrši pribavljnje operanada tj. na kojuse vrši upis.

Posmatrajmo strukturu podataka koja se vodi (book keeping). Navedene instruction tabela ne postoji kao deo hardvera ali je data radi lakšeg praćenja.

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F6,34(R2)			
LD F2,45(R3)			
MULD F0,F2,F4			
SUBD F8,F6,F2			
DIVD F10,F0,F6			
ADDD F6,F8,F2			

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	Yes	SUB	Mem(34+Regs(R2))			Load2
Add2	Yes	ADD			Add1	Load2
Add3	No					
Mult1	Yes	MULT		Regs(F4)	Load2	
Mult2	Yes	DIV		Mem(34+Regs(R2)	Mult1	

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	Mult1	Load2		Add2	Add1	Mult2					

Ove tri tabele prikazuju situaciju kada su sve instrukcije izdate. Ali je samo prva LOAD instrukcija završena i upisala je svoj rezultat na CDB. Vj i Vk polja nam kazuju vrednost operanda (na jeziku za opis hardvera). Load i Store buffer–i nisu prikazani. Load buffer2 je jedini zauzeti load bafer i to zauzet izvršavajući instrukciju 2 tj. učitavajući nešto sa adrese R3+45.

Treba zapamtiti da jedan operand u jednom trenutku ne može imati obeleženo i Q i V polje.

Dakle, F6 je dostupan pa je prva instrukcija kompletirana. F2 se još nije upisao na CDB (druga instrukcija još nije upisala rezultat). Ostale instrukcije su sve izdate. ADD je mogla biti izdata bez obzira na antizavisnosti.

Vidimo da se kod rezervacionih stanica nigde ne koriste imena registara jer su izvršena preimenovanja. Čim je dostupan rezultat, on se upisuje u rezervacionu stanicu pa ADDD može da krene u izvršenje.

Sada ćemo da krenemo u simulaciju (kao što smo simulirali Scoreboard). Dakle, ako usvojimo iste latencije instrukcija kao u Scoreboard–u dobićemo sledeće tabele.

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F6,34(R2)			
LD F2,45(R3)			
MULD F0,F2,F4			
SUBD F8,F6,F2			
DIVD F10,F0,F6			

ADDD F6,F8,F2			

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULT	Mem(45+Regs(R3))	Regs(F4)		
Mult2	Yes	DIV		Mem(34+Regs(R2)	Mult1	

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	Mult1					Mult2					

Jedino su MUL i DIV instrukcije ostale nedovršene. Ova situacija se razlikuje od one koju smo imali u Scoreboard–u po tome što smo eliminacijom WAR hazarda dozvolili ADDD instrukciji da završi svoje izračunavanje odmah posle SUBD instrukcije, od koje je inače zavisila.

DIVD je ostala u ISSUE fazi jer čeka rezultat instrukcije MULTD, a ADDD je okončala svoje izvršavanje (kod Scoreboarda ona nije mogla da uđe u EXECUTION fazu zbog antizavisnosti).

U sledećoj tabi su prikazani koraci kroz koje svaka instrukcja mora da prođe. LOAD i STORE su jedini izuzetci. LOAD se može izvršiti odmah čim na nju dođe red. Kada je izvršavanje završeno, i CDB je slobodan, LOAD stavlja svoj rezultat na CDB kao bilo koja druga Funkcionalna Jedinica. STORE naredbe dobijaju svoje vrdnosti sa CDB–a ili iz registrskog fajla i izvršava se nezavsino. Kada završe, onda setuju svoje BUSY polje na OFF da bi pokazale svoju spremnost za naredne instrukcije, baš kao i LOAD bafer ili rezervacione stanice.

Da bi smo shvatili sve prednosti eliminacije WAW i WAR hazarda dinamičkim preimenovanjem registara moramo se pozabaviti analizom sledećom petljom. Razmatraćemo sledeći niz naredbi za množenje niza skalarom koji je smešten u F_2.

Loop:	LD	$F_0(R_1)$
	MULTD	F_4,F_0,F_2
	SD	$0(R_1),F_4$
	SUBI	$R_1,R_1,\#8$
	BNEZ	R_1,Loop

Instruction status	Wait until	Action or Bookkeeping

ISSUE	Station or buffer empty	If (register(S1).Q_i <>0)
		Then (RS(r).Q_i <– Register(S1).Q_i)
		Else (RS(r).V_i <– S1); RS(r).Q_i <– 0);
		If (Register(S2).Q_i<>0
		Then (RS(r).Q_k <– Register(S2).Q_i);
		Else (RS(r).V_k <– S2; RS(r).Q_k<–0);
		RS(r).Busy <– Yes;
		Register (D).Q_i=r;

EXECUTE	(RS(r).Q_i=0) and (RS(r).Q_k=0)	None – operands are in V_i and V_k
WRITE RESULT	Execution comleted at r And CDB available	Za svako x (if Registe(x).Q_i=r) then (Fx <–result; Register(x).Q_i <– 0)); Za svako x (if RS(x).Q_i=r) then (RS(x).V_i <–result; RS(x).Q_i <– 0)); Za svako x (if RS(x).Q_k=r) then (RS(x).V_k <–result; RS(x).Q_k <– 0)); Za svako x (if Store(x).Q_i=r) then (Store(x).V <–result; RS(r).Busy <– No));

Ovom tabelom su prikazani koraci u algoritmu i slovi (zahtevi) šta je potrebno za svaki korak. Za izdavanje instrukcije D je odredišni registar a S1 i S2 su izvorišni registri (njihova imena odnosno brojevi) a r je rezervaciona stanica ili bafer za koji je povezan D registar. RS je struktura podataka rezervacione stanice. Vrednost koja se dobije iz rezervacione stanice ili iz load bafer se naziva result. Register je registarska struktura podataka a ne registarski fajl. Store je takođe struktura podataka store buffera.

Kada se izda instrukcija, u Qi polje se upisuje broj rezervacione stanice ili bafera u koju je instrukcija izdata. Ako su operandi dostupni u registrima, oni su smešteni u V poljima. U suprotnom (ukoliko nisu

dostupni u registrima) Q polja se setuju tako da označavaju rezervacionu stanicu koja će ih proizvesti. Instrukcija će čekati u rezervacionoj stanici sve dok joj oba operanda ne postanu dostupna. To da su oba operanda dostupna instrukcija vidi po tome što su oba Q polja prazna (setovana na nulu). Q polja se setuju na nulu i kada se instrukcija (tekuća) izda ili kada se instrukcija od koje tekuća zavisi završi i upiše svoj rezultat. Kada neka instrukcija završi svoje izvršavanje i kada njen rezultat postane raspoloživ na CDB–u tada može i da upiše svoj rezultat.

Kao što sam obajsnio Scoreboard tehniku na jednostavnom primeru, krećući se kroz cikluse takta, tako ću i sada to učiniti sa Tomasulovim algoritmom.

Na početku su sve tabele prazne:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)			
LD F_2,45(R_3)			
MULD F_0,F_2,F_4			
SUBD F_8,F_6,F_2			
DIVD F_{10},F_0,F_6			
ADDD F_6,F_8,F_2			

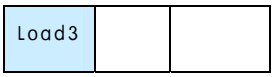
Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i											

	Busy	Adress
Load1		
Load2		


Load3

Idemo sada u ciklus 1:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1		
LD F_2,45(R_3)			
MULD F_0,F_2,F_4			
SUBD F_8,F_6,F_2			
DIVD F_{10},F_0,F_6			
ADDD F_6,F_8,F_2			

Prva Load instrukcija je izdata. Zato što postoji slobodna rezervaciona stanica.

	Busy	Adress
Load1	Yes	34+R_2
Load2	No	
Load3	No	

Load bafer1 postaje zauzet jer čita sa adrese 34+R_2.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Sve rezervacione stanice su slobodne (nisu zauzete).

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i				Load1							

Registar F_6 čeka na rezultat iz Load bafera 1.

Idemo sada u ciklus 2:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1		
LD F_2,45(R_3)	2		
MULD F_0,F_2,F_4			
SUBD F_8,F_6,F_2			
DIVD F_{10},F_0,F_6			
ADDD F_6,F_8,F_2			

U ovom ciklusu se i druga LOAD instrukcija izdaje, dok prva čeka svoju latenciju da bude izvršena.

	Busy	Adress
Load1	Yes	34+R_2
Load2	Yes	45+R_3
Load3	No	

Load bafer2 postaje zauzet jer čita sa adrese 34+R_2.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	No					

Sve rezervacione stanice su slobodne (nisu zauzete) idalje.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i		Laod2		Load1							

Registar F_6 čeka na rezultat iz Load bafera idalje, ali i registar F_2 sada čeka na rezultat iz funkcionalne load bafera 2.

Idemo sada u ciklus 3:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	F_6,34(R_2)	1	3	
LD	F_2,45(R_3)	2		
MULD	F_0,F_2,F_4	3		
SUBD	F_8,F_6,F_2			
DIVD	F_{10},F_0,F_6			
ADDD	F_6,F_8,F_2			

Pošto je izdata u ciklusu 1, prva load naredba se završava u ciklusu 3 (latencija = 1).

Druga load naredba i dalje čeka.

Izdata je i MULD naredba, zato što ima slobodne rezervacione stanice.

	Busy	Adress
Load1	Yes	34+R_2
Load2	Yes	45+R_3
Load3	No	

Naravno, oba bafera su još uvek zauzeta (sve dok ne upišu rezultat u odredišne registre).

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULTD		R(F_4)	Load2	
Mult2	No					

V_j, koji je prazan govori da izvorišni operand nije dostupan, a V_k govori da je drugi izvorišni operand dostupan i da je njegova vrednost R(F_4).

Polje Q_j kazuje koju funkcionalnu jedinicu ili bafer čeka prvi operand. U našem slučaju to je bafer load2 koji treba da upiše u registar F_2. Drugi operand je dostupan u registru F_4.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	Laod2		Load1							

Sada i F_0 čeka na nešto. U našem slučaju, na Mult1 funkcionalnu jedinicu.

Idemo sada u ciklus 4:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1	3	4
LD F_2,45(R_3)	2	4	
MULD F_0,F_2,F_4	3		
SUBD F_8,F_6,F_2	4		
DIVD F_{10},F_0,F_6			
ADDD F_6,F_8,F_2			

Sada je prva LOAD instrukcija već došla do upisa rezultata i to je i učinila.

Druga load naredba se izvršila, ostaje samo da upiše svoj rezultat.

MULD čeka svoju latenciju.

Izdata je i naredba SUBD jer ima mesta u rezervacionoj stanici za takvu naredbu.

.

	Busy	Adress
Load1	No	
Load2	Yes	45+R_3
Load3	No	

Ostaje samo još da se druga load naredba okonča pa da oba load bafera budu slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	Yes	SUBD	M(A_1)			Load2
Add2	No					
Add3	No					
Mult1	Yes	MULTD		R(F_4)	Load2	
Mult2	No					

Sada je i add1 rezervaciona stanica zauzeta naeredbom SUBD. Njeni operandi su nedostupni, tačnije, drugi operand je nedostupan jer ga treba upisati laod2 bafer. Prvi je dostupan i on je F_6. njega smo preimenovali u M(A_1) zato što njega stvara jedna od predhodnih naredbi. VALJDA JE ZATO.

Što se tiče naredbe MULD ona je i dalje blokirana jer čeka load2 bafer.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	Laod2		$M(A_1)$	Add1						

F_6 registar treba da upiše u lokaciju $M(A_1)$. Sve ostalo je jasno zar ne?

Idemo sada u ciklus 5:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	$F_6,34(R_2)$	1	3	4
LD	$F_2,45(R_3)$	2	4	5
MULD	F_0,F_2,F_4	3		
SUBD	F_8,F_6,F_2	4		
DIVD	F_{10},F_0,F_6	5		
ADDD	F_6,F_8,F_2			

Druga LOAD naredba je sada okončala i upisala rezultat.

Muld naredba čeka svoju latenciju

Subd takođe

DIVD je izdata jer postoji slobodna rezervaciona stanica.

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_i	V_k	QJ	Q_k
Add1	Yes	SUBD	$M(A_1)$	$M(A_2)$		
Add2	No					
Add3	No					
Mult1	Yes	MULTD	$M(A_2)$	$R(F_4)$		
Mult2	Yes	DIVD		$M(A_1)$	Mult1	

Sada je završena load2 instrukcija i generisala nam je rezultat koji će iskoristiti SUBD naredba. Narvno, i on se preimenuje jer će i on kasnije biti ponovo korišćen. Sada SUBD ima dostupne operande.

I MULD je čekala na load2 bafer, tako da sada i ona ima slobodne operande. Ovde koristimo isto ono imenovanje, nema potrebe da ponovo preimenujemo.

DIVD je izdata i čeka MULD da joj napravi izvorišni operand.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	Mult1	$M(A_2)$			Add1	Mult2					

Idemo sada u ciklus 6:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1	3	4
LD F_2,45(R_3)	2	4	5
MULD F_0,F_2,F_4	3		
SUBD F_8,F_6,F_2	4		
DIVD F_{10},F_0,F_6	5		
ADDD F_6,F_8,F_2	6		

Multd čeka svoju latenciju

Subd takođe

Divd takođe

ADDD je izdata, jer postoji slobodna rezervaciona stanica.

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	Yes	SUBD	M(A_1)	M(A_2)		
Add2	Yes	ADDD		M(A_2)	Add1	
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

U rezervacionu stanicu ušla je ADDD jer postoji jedna add slobodna funkcionalna jedinica (add2).

Sve ostalo je ostalo nepromenjeno.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	$M(A_2)$		Add2	Add1	Mult2					

Idemo sada u ciklus 7:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1	3	4
LD F_2,45(R_3)	2	4	5
MULD F_0,F_2,F_4	3		
SUBD F_8,F_6,F_2	4	7	
DIVD F_{10},F_0,F_6	5		
ADDD F_6,F_8,F_2	6		

Multd čeka svoju latenciju

Subd je završio (latencija mu je bila 2)

Divd takođe čeka

ADDD takođe čeka

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	Yes	SUBD	M(A_1)	M(A_2)		
Add2	Yes	ADDD		M(A_2)	Add1	
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

Sve ostalo je ostalo nepromenjeno.

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	$M(A_2)$		Add2	Add1	Mult2					

Skočimo sada u ciklus 10:

Instruction Status:

Instruction	ISSUE	EXECUTION	WRITE RESULT
LD F_6,34(R_2)	1	3	4
LD F_2,45(R_3)	2	4	5
MULD F_0,F_2,F_4	3		
SUBD F_8,F_6,F_2	4	7	8
DIVD F_{10},F_0,F_6	5		
ADDD F_6,F_8,F_2	6	10	

Multd čeka svoju latenciju

Divd takođe čeka

ADDD je krenula u izvršenje jer je premostila iz exe faze operand koji joj je falio. Dakle, još u sedmom ciklusu premošćenje je obavljeno, od tog trenutka treba da prođe latencija ADD naredbe koja je 2 kloka i da sada, u desetom, imamo završetak.

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_i	V_k	QJ	Q_k
Add1	No					
Add2	Yes	ADDD	(M–M)	M(A_2)		
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	$M(A_2)$		Add2	Add1	Mult2					

Skočimo sada u ciklus 11:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	F_6,34(R_2)	1	3	4
LD	F_2,45(R_3)	2	4	5
MULD	F_0,F_2,F_4	3		
SUBD	F_8,F_6,F_2	4	7	8
DIVD	F_{10},F_0,F_6	5		
ADDD	F_6,F_8,F_2	6	10	11

Multd čeka svoju latenciju

Divd takođe čeka

ADDD je Upisala svoj rezultat i time okončala svoj životni ciklus. Da li je ona smela da upiše svoj rezultat s obzirom da F_6 treba tek da bude pročitano od strane DIV naredbe. Naravno da sme zato što se vrši preimenovanje.

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_i	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	$M(A_2)$		(M–M+M)	(M–M)	Mult2					

Skočimo sada u ciklus 15:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	F_6,34(R_2)	1	3	4
LD	F_2,45(R_3)	2	4	5
MULD	F_0,F_2,F_4	3	15	
SUBD	F_8,F_6,F_2	4	7	8
DIVD	F_{10},F_0,F_6	5		
ADDD	F_6,F_8,F_2	6	10	11

Muld je dočekala svoje izvršenje. Njena latencija je bila 10, ali su joj operandi bili dostupni tek u petom ciklusu takta.

Sve ostalo je isto !

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_i	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	Mult1	M(A_2)		(M–M+M	(M–M)	Mult2					

Skočimo sada u ciklus 15:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	F_6,34(R_2)	1	3	4
LD	F_2,45(R_3)	2	4	5
MULD	F_0,F_2,F_4	3	15	
SUBD	F_8,F_6,F_2	4	7	8
DIVD	F_{10},F_0,F_6	5		
ADDD	F_6,F_8,F_2	6	10	11

Muld je dočekala svoje izvršenje. Njena latencija je bila 10, ali su joj operandi bili dostupni tek u petom ciklusu takta.

Sve ostalo je isto !

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	Yes	MULTD	M(A_2)	R(F_4)		
Mult2	Yes	DIVD		M(A_1)	Mult1	

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30

Q_i	Mult1	$M(A_2)$		(M–M+M	(M–M)	Mult2					

Skočimo sada u ciklus 56:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	$F_6,34(R_2)$	1	3	4
LD	$F_2,45(R_3)$	2	4	5
MULD	F_0,F_2,F_4	3	15	16
SUBD	F_8,F_6,F_2	4	7	8
DIVD	F_{10},F_0,F_6	5	56	
ADDD	F_6,F_8,F_2	6	10	11

Divd je sada završio izvršavanja. Njegova latencija je bila 40 ciklusa takta, ali su mu operandi bili dostupni tek u 16–tom ciklusu takta pa se on tek sada završio.

Sve ostalo je isto !

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	Yes	DIVD	$M*F_4$	$M(A_1)$		

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	$M*F_4$	$M(A_2)$		(M–M+M	(M–M)	Mult2					

Skočimo sada u ciklus 57:

Instruction Status:

Instruction		ISSUE	EXECUTION	WRITE RESULT
LD	F_6,34(R_2)	1	3	4
LD	F_2,45(R_3)	2	4	5
MULD	F_0,F_2,F_4	3	15	16
SUBD	F_8,F_6,F_2	4	7	8
DIVD	F_{10},F_0,F_6	5	56	57
ADDD	F_6,F_8,F_2	6	10	11

Divd je sada upisao svoj rezultat.

Sve ostalo je isto !

.

	Busy	Adress
Load1	No	
Load2	No	
Load3	No	

Oba load bafera su slobodna.

Reservation stations:

Name	Busy	Op	V_j	V_k	QJ	Q_k
Add1	No					
Add2	No					
Add3	No					
Mult1	No					
Mult2	Yes	DIVD	M*F_4	M(A_1)		

Register status:

Field	F0	F2	F4	F6	F8	F10	F12	F14	F16	.....	F30
Q_i	M*F_4	M(A_2)		(M–M+M	(M–M)	Mult2					

Glavni nedostaci Tomasulovog algoritma su:

- Kompleksnost
- Mnoga asocijativna traženja pri velikim brzinama
- Performanse su ograničene CDB–om

Dinamička predikcija grananja

Upoznali smo se sa **hardverskim** tehnikama za otklanjanje zastoja koji su posledica hazarda po podacima (Scoreboard i Tomasulov algoritam). Učestanost naredbi grananja i naredbi bezuslovnog skoka (JUMP) zahteva da se napadnu i potencijalni zastoji uzrokovani **kontrolnim hazardima**. Zaista, kako raste količina ILP–a koji želimo da eksploatišemo, tako kontrolni hazardi postaju sve veći ograničavajući faktor. Proučili smo nekoliko statičkih šema za rešavanje problema uzrokovanih naredbama grananja (**zakašljeno grananje**, **odmotavanje petlje**). Ove šeme su statičke jer preduzete akcije ne zavise od dinamičkog ponašanja programa.

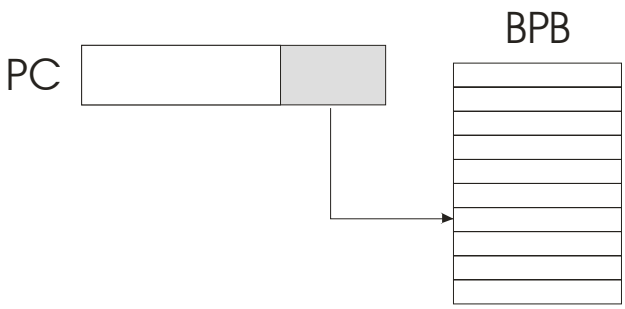
Sada ćemo se upoznati sa hardverskim tehnikama za **dinamičko predviđanje** ishoda grananja. Predviđanje će se promeniti ako grananje promeni svoje ponašanje u toku izvršenja programa. Cilj svih tehnika je da omoguće procesoru da razreši ishod grananja ranije i tako spreči da kontrolne zavisnosti izazivaju zastoje. Efikasnost tehnika za predviđanje grananja ne zavisi samo od pouzdanosti predviđanja, već i od cene grananja kada je predviđanje tačno i cene kada je ono netačno.

Dakle, predviđanje se menja u zavisnosti od prethodnog ponašanja naredbi grananja u toku izvršenja programa. Nastoji se da se uklone zastoji i da se što pre počne sa izvršenjem sledeće naredbe. Ta sledeća naredba može da bude ili naredba koja sekvencijalno sledi posle naredbe grananja (ukoliko do grananja nije došlo) ili neka naredba koja je specificirana u samoj naredbi grananja (ukoliko je do grananja došlo). Predviđanje ne zavisi samo od pouzdanosti već i od cene (kolika je cena kad je tačno predviđanje, a kolika kada je netačno)

Branch Prediction Buffer BPB

Najjednostavnija tehnika za dinamičko predviđanje grananja zove se **BRANCH PREDICTION BUFFER** (BDP – bafer perdikcije grananja) ili **HISTORY TABLE** (tabela istorije grananja).

BPB je u suštini mala memorija koja je indeksirana (adresirana) niskim bitovima adrese naredbe grananja (neke BRANCH instrukcije).



Jedan element BPB–a je u stvari samo 1 bit koji govori o predviđanju. **Nula** označava da se grananje neće obaviti a jedinica označava da će do grananja doći.

Ovo je najjednostavnija vrsta bafera i od koristi je samo kada je kašnjenje (uzrokovano grananjem) veće od vremena potrebnog da se izračuna moguća ciljna vrednost programskog brojača tj. potencijalna adresa grananja. Drugim rečima, kada je adresa grananja poznata ranije od uslova grananja.

U BPB–u imamo dakle po jedan bit za svaku naredbu grananja. Taj bit u stvari kaže da li se poslednji put grananje obavilo ili ne, a na osnovu toga se vrši predviđanje ishoda narednog grananja te iste instrukcije grananja.

Primećujemo da ovo nije ništa drugo nego uvođenje jednog od osnovnih zakona prirode, a to je tromost ili **inercija**. Ukoliko se poslednji put dogodio grananje, mi predviđamo da će se ponovo dogoditi, jer uvodimo inerciju. (inercija je težnja tela da zadrži svoje trenutno stanje).

Pošto neke naredbe mogu da imaju iste zadnje bitove, a da se razlikuju samo u višim bitovima, onda predviđanje koje se nađe u BPB i nije baš 100% sigurno za tu naredbu (koristi se samo nekoliko nižih bitova, po može da se negde poklopi sa nekom drugom naredebom).

Bez obzira na to šta se desilo (da li je to ta naredba ili ne) u BPB–u uzima se kao da je baš za nju i počinje se sa pribavljanjem instrukcije na osnovu predviđanja. Ukoliko je **jedinica** pribavlja se naredba specificirana u naredbi grananja, a ukoliko je **nula** pribavlja se sekvencijalno naredna naredba. Ako se ispostavi da je predviđanje bilo pogrešno, predikcioni bit se **invertuje**.

Efikasnost ove tehnike zavisi od toga koliko često je reč o BRANCH instrukciji koja je od interesa (da se slučajno nisu poklopile adrese –bar što se nižih delova adrese tiče) i koliko je predviđanje tačno !

Za naš petostepeni protočni sistem (sa modifikacijom). Uslov i adresa grananja su poznati u ID fazi i ovaj prikaz tu ne bi dao nikakvo poboljšanje. Korišćenje **BPB–a** ima smisla ako je adresa grananja poznata pre ID faze. BPB–u se pristupa u ID fazi. (prediskcioni bit se stalno menja).

NPR:

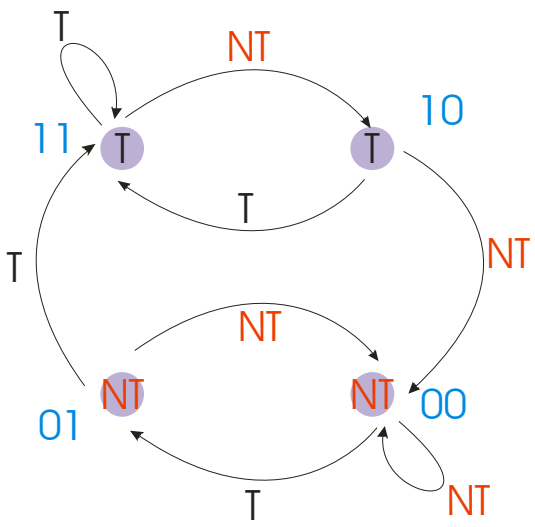
BPB daje dobre rezultate kod visokoregularnih predviđenih grananja NPR za petlje koje se vrte. Ako petlju vrtimo 10 puta, rezultat grananja će samo jedared biti ne, a 9 puta da. Kolika je pouzdanost predviđanja za ovo grananje pod pretpostavkom da predikcioni bit za ovo grananje ostaje u BPB–u. jedina greška se javlja na kraju petlje. To znači da je ovde predviđanje bilo itekako **efikasno**. Koliko je to Efikasno ? Predviđanje će biti tačno uvek sem kod prve i poslednje iteracije petlje. Pogrešno predviđanje kod poslednje iteracije petlje je neizbežno jer će bit biti postavljen na 1 (grananje treba da se obavi) pošto se grananje obvavilo 9 puta za redom do tada. Ali pošto je petlja završena, grananja više nema. Što se tiče pogrešnog predviđanja kod prve iteracije, ono nastaje jer je bit bio postavljen na 0 predhodnim izvršenjem poslednje iteracije prethodne petlje. Dakle, možemo da zaključimo da je efikasnost u 90% slučajeva negde oko 80%. U idealnom slučaju pouzdanost predviđanja će se poklopiti sa učestanošću *taken* branch. (100%).

BPB–u se pristupa u ID fazi – kada je instrukcija dekodirana i kada znamo da je instrukcija grananja (ako bi smo BPB–u pristupali za neku instrukciju koja nije grananje, mogli bi da se odgranamo ko zna gde, a to naravno nevalja.

Ovaj **jednobitni BPB** je fiksan ako imamo grananja koja se ponašaju predvidljivo (kao kod petlje na primer). To su takozvana visokoregularna grananja. Kod visokoregularnih grananja pomoću BPB–a ćemo dva puta pogrešiti da pretpostavimo grananje u petlji ali će ostala predviđanja biti tačna.

Predikcioni bit se menja ako grananje promeni ponašanje, kako bi označio poslednje grananje. Dakle, nameće se rešenje da se na neki način poveća tromost BPB–a.

Da bi se poboljšale performanse BPB–a koriste se **2–bitni prediktori**. Za 2 bitne se predviđanje menja tek ako ono dva puta uzastopno bude pogrešno. Graf promene stanja za 2 bitni prediktor dat je na slici:



M bitni prediktor je u suštini brojač od 0 do 2^m–1. Ovaj brojač je zasićen brojač što znači da ako dođe do 2^m–1 i ponovo se inkrementira, on tu i ostaje. Takođe, ukoliko se nađe na nuli a pokuša de se dekrementira, on ostaje na nuli i dalje. Takozvani **saturated counter.**

Vrednosti brojača >= 2^{m-1} (što je tačno polovina intervala) predstavljaju akciju **T (Taken)** što znači da će se grananje obaviti.

Vrednosti brojača < 2^{m-1} (što je tačno polovina intervala) predstavljaju akciju **NT (Not Taken)**, znači grananje se neće obaviti.

Svaki put kada se grananje obavi, inkrementira se brojač. Naravno, važi i obrnuto, kada se grananje ne obavi dekrementira se. 2–bitni predisktori se ponašaju skoro isto tako dobro kao i M bitni prediktori, tako da se uglavnom koriste 2 bitni jer su jeftiniji i prostiji.

Predikcioni baferi se mogu implementirati kao mali, specijalni, keš kome se pristupa na osnovu adrese instrukcije u toku IF faze. Ako je dekodirana BRANCH instrukcija, i ako je predviđanje da će se ona obaviti, pribavljanje nove instrukcije počinje sa ciljne adrese grananja, čim je poznat sadržaj PC–a. Ukoliko se pak predviđa da do grananja neće doći sledeća naredba koja će se pribaviti je sekvencijalno sledeća naredbi grananja.

Evidentno je da ova šema nije od koristi kod 5–stepenog protočnog sitema (naš primer) jer su uslov grananja i adresa grananja poznati u istom trenutku. Nešto kasnije videćemo koja šema može da poboljša karakteristike našeg protočnog sistema, DLX).

Na testovima obavljenim sa programima SPEC89 i SPEC92 dobijene su informacije da se sa veličinom BPB–a od oko 4 kb postiže pouzdanost predviđanja od oko 82–99%. BPB veličine 4kb se ponaša isto kao i bafer beskonačne veličine. Na osnovu testova, pokazalo se da INT programi imaju dosta teško predvidiva grananja. Da bi se povećala pouzdanost predviđanja može se:

- Povećati BPB (ali samo do 4 Kb)
- Poboljšati šema koja se koristi za predviđanje.

Pokazalo se da je bafer veličine 4k dovoljno velik i da se ponaša približno kao bafer beskonačne veližine. Ovo jasno govori da broj pogodaka (hit rate) nije ograničavajući faktor. Takođe, kao što smo već napomenuli povećanje broja bitova predikcije ima mali uticaj na efikasnost prediktora. Dakle, moramo razmislitu u samoj šemi po kojoj mi zaključujemo da li će do grananja doći ili ne.

Korelacioni prediktori

Dosadašnja tehnika je koristila za predviđanje grananja ponašanje te iste naredbe grananja pre toga. Da bi se poboljšala pouzdanost za tekuću naredbu grananja, moguće je koristiti i neke prethodne naredbe koje nisu baš te naredbe grananja (ali jesu naredbe grananja ili su tesno povezane sa njima). Dakle, u predikciji grananja učestvuju i naredbe koje su različitog tipa od onih za koje vršimo predikciju.

Primer iz SPEC92:

Prevedimo ovaj program na nekom pseudokodu na asemblerski jezik. Neka je u R_1 smesteno aa a u R_2 smesteno bb:

IF	(aa==2)
	aa=0
IF	(bb=2)
	bb=0
If	(aa!=bb)
	( )

	SUBI	R3,R1,#2	
	BNEZ	R3,L1	B1
	ADD	R1,R0,R0	
L1	SUBI	R3,R2,#2	
	BNEZ	R3,L2	B2
	ADD	R2,R0,R0	

L2	SUB	R3,R1,R2	
	BEQZ	R3,L3	B3
L3	.....	.....	

B1 se obavlja ako je aa <> 2

B2 se obavlja ako je bb <> 2

B3 se obavlja ako je aa = bb

Dakle, kad god se B1 i B2 ne obave B3 se obavlja. Ovu osobinu nisu iskorišćavali prethodni prediktori o kojima je bilo reči. Dakle, oni nisu koristili ponašanje prethodnih naredbi grananja u cilju predikcije tekuće naredbe. Ovi drugi prediktori će ove pogodnosti itekako eksploatisati. Prediktori koji ovo koriste zovu se **korelacioni prediktori** ili **dvo–nivoiski prediktori**). Da bi videli kako ti prediktori rade uzmimo jedan jednostavan primer (izabran u ilustrativne svrhe).

Primer:

IF	(d==0)		BNEZ	R1,L1	B1
	D=1		ADDI	R1,R1,#1	
IF	(d=1)	L1	SUBI	R3,R1,#1	
	( )		BNEZ	R3,L2	B2

Grananja koja odgovaraju dvema IF naredbama označena su sa B1 i B2. Da vidimo kako bi se odvijalo izvršenje ovog kodnog segmenta ako bi d inicijalno imalo vrednosti 0,1,2

Polazno d	D==0 ?	Ponašanje grananja b1	Vrednost d pre b2	d==1?	Ponašanje grananja b2
0	DA	NT	1	DA	NT
1	NE	T	1	DA	NT
2	NE	T	2	NE	T

Iz tabele se vidi da ako je B1 = NT tada je B2 sigurno NT. **Korelacioni prediktor** može iskoristiti ovu osobinu i time steći prednost u odnosu na **standardni prediktor**. Dakle, ako se B1 ne obavlja, onda će se i B2 sigurno ne obavlja. Dakle, Za B1 = NT sigurno je i B2 = NT

U sledećoj tablici je dato ponašanje 1 bitnog prediktora kada d uzima naizmenično vrednosti 0 i 2.... Treba napomenuti da su predviđanja **inicijalno** podešena na **No Taken** (dakle predpostavlj se da do grananja neće doći).

Primer:

d?	Predikcija za b_1	Akcija b_1	Nova predikcija za b_1	Predikcija za b_2	Akcija za b_2	Nova predikcijab_2

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

Primećujemo da je tačnost pogađanja 0%. Razmotrimo sada prediktor koji koristi **1 bit korelacije**. Najlakši način da ovo razumemo je da zamilimo da svako grananje ima dva posebna predikciona bita: jedno predviđanje se koristi ako se poslednje grananje nije obavilo, a drugo predviđanje se koristi ako se poslednje grananje obavilo. Uočimo da, u opštem slučaju, poslednja naredba grananja koja je izvršena nije ista naredba grananja za koju je obavljeno predviđanje (naravno može se odnositi i na tu naredbu ako druge naredbe uopšte nema). Par predikcionih bitova se beleži zajedno, pri čemu prvi bit prediktora predviđa ako se poslednje grananje nije obavilo, a drugi bit predviđanje ako se poslednje grananje obavilo.

Dakle, kod korelacionog 1–bitnog prediktora ima se utisak da postoje po 2 bita za svaku instrukciju grananja. Prvi bit (x) predstavlja predviđanje pod uslovom da poslednje grananje nije nije bilo obavljeno, a drugi bit (y) predviđanje koje se koristi ako je poslednje grananje bilo obavljeno. Poslednja naredba grananja koja je izvršena ne mora biti ta ista naredba za koju se vrši predviđanje (ali može ako se vrti pelja). Taj par predikcionih bitova se beleži zajedno u BPB–u i koristi u zavisnosti od akcije prethodne instrukcije grananja.

X,Y	Predikcija ako poslednje Grananje nije obavljeno	Predikcija ako je poslednje Grananje obavljeno
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

Da vidimo sda kakvo je ponašanje 1–bitnog prediktora sa 1 bitom korelacije, kada su prediktori inicijalizovani na NT/NT.

Ponašanje 1–bitnog prediktora (za T/NT se koristi 1b) sa jednim bitom korelacije (uzima se ponašanje jedne prethodne instrukcije grananja) dato je u tablici za d= 2,0,2,0

D=?	Predikcija za b_1	Akcija	Nova b_1 predikcija	Predikcija za b_2	Akcija	Nova b_2 predikcija
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

U ovom slučaju pogrešno predviđanje je jedino u prvoj iteraciji, kada je d=2. korektno predviđanje za B1 je zbog izbora vrednosti za d, jer B1 očigledno ne zavisi od B2. Međutim, korektno predviđanje za B2 je posledica korelacije.

Uzmimo da su u startu predikcije postavljene na NT/NT i da poslednje grananje nije obavljeno. Za B2 je predviđanje dobro ako se B1 obavi. Ovo je **(1,1)** prediktor (izgleda kao da je u pitanju 2–bitni prediktor, ali su značenja bitova drugačija). Prvi bit koristi istoriju od jedne instrukcije, drugi bit se pak koristi za predikciju. Čak da smo odabrali i druge vrednosti za d, prdikcija za B2 bi uvek bila tačna kada je B1 = NT.

Ovaj prediktor se zove **(1,1) prediktor** jer koristi ponašanje prethodne naredbe grananja da odabere jedan 1–bitni prediktor iz para prediktora.

U opštem slučaju (m,n) prediktor koristi ponašanje m poslednjih naredbi grananja, da odabere jedan od 2^m n–bitnih prediktora.

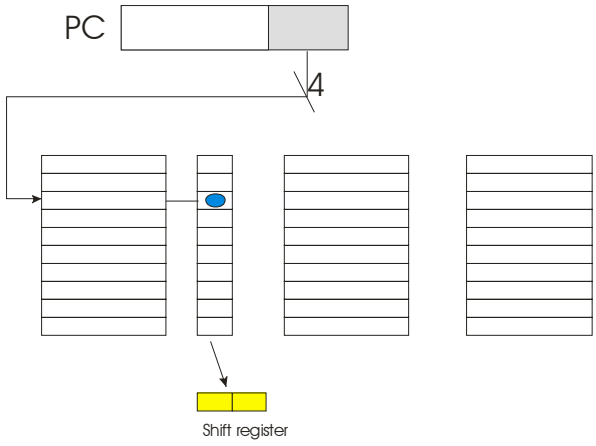
Korelacioni prediktori imaju veću pouzdanost predviđanja od 2–bitnih prediktora, a zahtevaju mali (trivijalni) dodatni hardver. Naime, globalna istorija m poslednjih grananja se može zapamtiti u m–bitnom **šift registru**, pri čemu svaki bit beleži da li je granaje bilo obavljeno ili ne (T ili NT).

Kao što smo za 2–bitni prediktor kazali da je specijalni slučaj m–bitnog, tako i za ovaj možemo reći da je ustvari samo specijalan slučaj (m,n) prediktora. Kod (m,n) prediktora se koristi ponašanje m predhodnih instrukcija grananja, pri čemu se koristi n bitova za predviđanje grananja. U ovom slučaju bafer ima 2^m ulaza sa po n bitova za svaki ulaz.

Ponašanje m poslednjih instrukcija se može pamtiti u jednom shift registru. Poslednji bit u registru označava ponašanje poslednje instrukcije grananja. Ovo je (verovatno ste primetili) jako jednostavan hardver.

Bafer za predikciju grananja semože adresirati korišćenjem konkatenacije nižih bitova adrese naredbe grananja i m–bitne globalne istorije (m–bitnog šift registra).

Recimo da imamo (2,2) prediktor i da se za selekciju ulaza u BPB koriste 4b instrukcije grananja. Imali bi smo 64 mogućnosti da izaberemo.



U PC–u je Adresa naredbe grananja čije ponašanje predviđamo (2^4=16).

Za selekciju se koristi konkatenacijaovih 4b sa šift registrom od 2b (dakle, dobijamo 2^{4+2}=64) pa zato imamo 64 ulaza.

Treba naponenuti da bafer nije dvodimenzionalan već je samo ovako predstavljen slikovito.

Šift registar služi da selektuje odgovarajuću kolonu.

U svakom elementu BPB–a su 2 bita (2 bita predikcije za grananje x)

Da bi smo uporedili šemu koja koristi korelaciju i šemu koja ne koristi korelaciju, kao kriterijum koristimo količinu informacija koja se pamti u BPB–u (koliko je bitova stanja).

Broj bitova stanja jednog (m,n) prediktora se dobija kao 2^m x n x (broj ulaza selektovanih nižim bitovima naredbe grananja. 2–bitni prediktor koji ne koristi korelaciju može da se posmatra kao korelacioni prediktor tipa (0,2). Ovaj prediktor sa 4k ulaza u baferu pamti 2^0 x 2 x 4k = 8k bitova. A (2,2) prediktor sa 4b adrese naredbe grananja 2^2 x 2 x 2^4=128 bitova.

Na osnovu ovoga možemo porediti ove šeme.

Nas zanima koliko ulaza ima predikcioni bafer koji koristi korelaciju a pamti isto bitova kao i onaj koji ne koristi korelaciju.Ako dobijemo manje ulaza, on je bolji zar ne? (lakši za proizvodnju).

2^2 x 2 b_2_ulaza = 8k b_2_ulaa je između 1k i 4k

korelacioni ima samo 1k ulaza.

Moguće je da se selekcija ulaza vrši samo na osnovu predhodne istorije, a ne na osnovu adrese grananja. Takav je (12,2) bafer – degenerisani slučaj. On ima 8k ulaza.

Jedna druga šema može omogućiti da se već na kraju IF faze dobije rezultat grananja (i može se iskoristiti u našem protočnom sistemu i, što je bitnije, dati poboljšanje).

U tom slučaju se koristi **BRANCH TARGET BUFFER** (bafer ciljne adresegrananja).

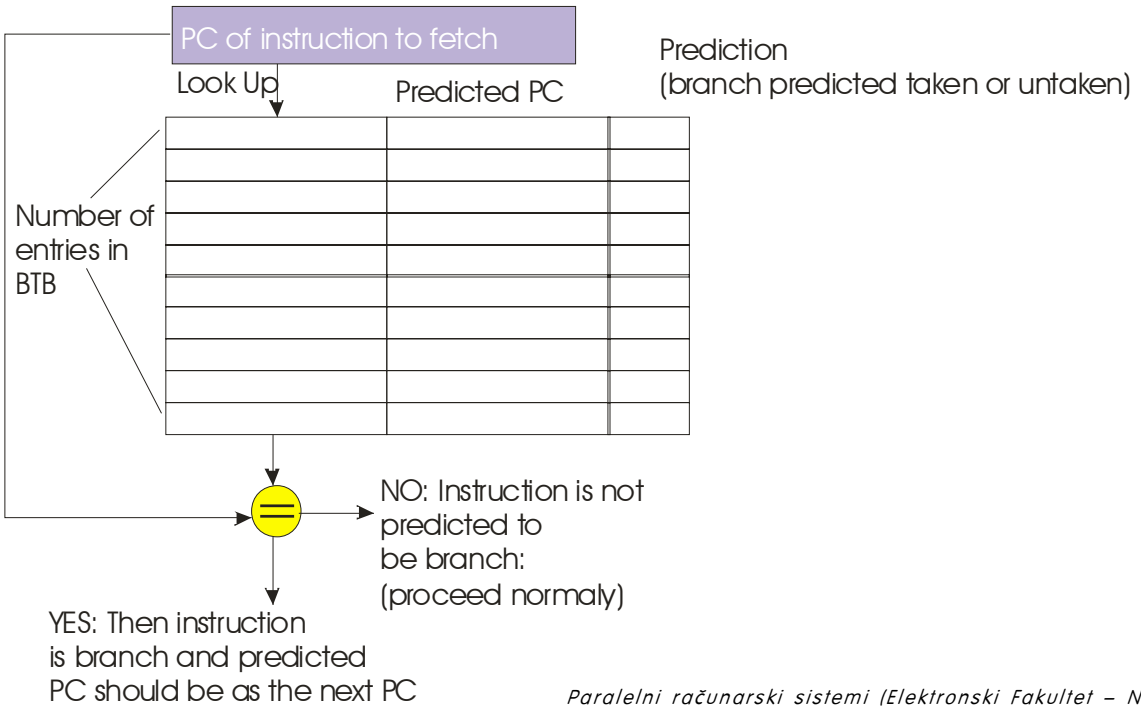
BRANCH TARGET BUFFER:

I ova tehnika je hardverska tehnika, koja redukuje kašnjenje izazvano BRANCH instrukcijama. Ovde se BTB–u pristupa u IF fazi i to kada još ne znamo da je u pitanju naredba grananja. Da bi se selektovao ulaz u BTB koristi se cela adresa naredbe skoka. Da bi se redukovalo kašnjenje zbog naredbe grannja potrebno je da znamo sa koje adrese treba pribaviti instrukciju na kraju IF faze. To znači da moramo da znamo d ali je još nedekodirana instrukcija instrukcija grananja, i ako jeste, koji je sledeći sadržaj programskog brojača (ishod).

Ako je u pitanju naredba grananja, i ako znamo koji je sledeći sadržaj PC, sveli bismo gubitke na 0. Bafer koji pamti predviđene adrese za sledeću instrukciju nakon naredbe grananja, zove se branch–target–bafer (BTB) ili branch–target–cache.

BPB – se pristupa u toku ID faze i tada je poznata ciljna adresa i predikcija.

Kada je upitanju BTB, baferu se pristupa u toku IF faze korišćenjem adrese pribavljene instrukcije, mogući branch, da bi se pristupilo bafweru. Ako imamo pogodak, onda znamo adresu sledeće instrukcije na kraju IF faze, što je 1 clk pre nego kod BPB–a.



Pošto je u pitanju predviđanje adrese sledeće instrukcije, pre dekodiranja instrukcije, mi moramo znati da li je pribavljena instrukcija procenjena kao **branch taken**.

BTB u prvoj koloni sadrži adrese naredbi grananja koje su već izvršene. U drugoj koloni su ciljne adrese (ako se grananje ostvari) i u trećoj koloni se nalazi predikcija za to grananje.

U PC–u se nalazi adresa pribavljene instrukcije. U ovoj **look up** tabeli se proverava da li je to instrukcija grananja ili ne? Ako se ustanov da to nije instrukcija grananja (nema je u tabeli) onda se ide dalje, a ako je ima, onda se na osnovu predikcije pribavlja sledeća naredba.

Dakle, ako se adresa pribavljene instrukcije upari sa nekom adresom u baferu, tada se odgovarajući predviđeni PC koristi kao sledeću PC. (Hardver za BTB je u suštini identičan hardveru za keš). Ako je ostvareno uparivanje pribavljanje nove instrukcije počinje odmah sa predviđene adrese u BTB–u.

Primetimo da se za razliku od BPB–a cela adresa mora poklopiti sa adresom u bafeu, a ne samo niži bajtovi adrese, jer će predviđeni PC postati novi PC pre nego što uopšte budemo znali da li je u pitanju naredba grananja.

Ako ne bi postojalo potpuno uparivanje adresa, onda bi predviđeni PC postao novi PC i za naredbe koje nisu naredbe grananja, to bi bilo katastrofalno.

U suštini, u BTB je potrebno staviti samo grananja koja su predviđena kao take, jer not taken grananja slede istu strategiju kao i instrukcije koje nisu naredbe grananja.

BTB može da bude i bez predikcije tj. bez treće kolone, tada se u BTB upisuju samo naredbe grananja za koje se predviđa da će se obaviti (one duge, se tretiraju kao da i nisu naredbe grananja, jer se njihov ishod unapred zna (prelazi se na sekvencijalno sledeću naredbu).

Komplikacije nastaju kada se koristi 2–bitni prediktor, jer se zahteva da se zapamte i informacije za teken i za not taken grananja. Jedn način da se ovo reši je rešenje koje se koristi kod Power PC–620.

Naime, usvajamo da bafer sadrži PC–relativne adrese uslovnog grananja, što čini ciljnu adresu konstantnom. Na slici koja je data dole prikazani su koraci kroz koje se prolazi kada se koristi BTB i gdese oni dešavaju u protočnom sistemu (tj. u kojim fazama).

Ako koristimo BTB, već na kraju IF faze je poznata adresa sledeće naredbe, tako da nemamo kašnjenje (ni od jednog ciklusa takta) naravno, ako je predviđanje bilo dobro. Da li je predviđanje bilo dobo saznajemo nakon ID faze. Ako predviđanje nije bilo dobro moramo da obrišemo prethodno pribavljenu naredbu.

Može da se desi da je u PC–u bila naredba grananja ali da ona nije bila u BTB–u. U tom slučaju treba ažurirati BTB.

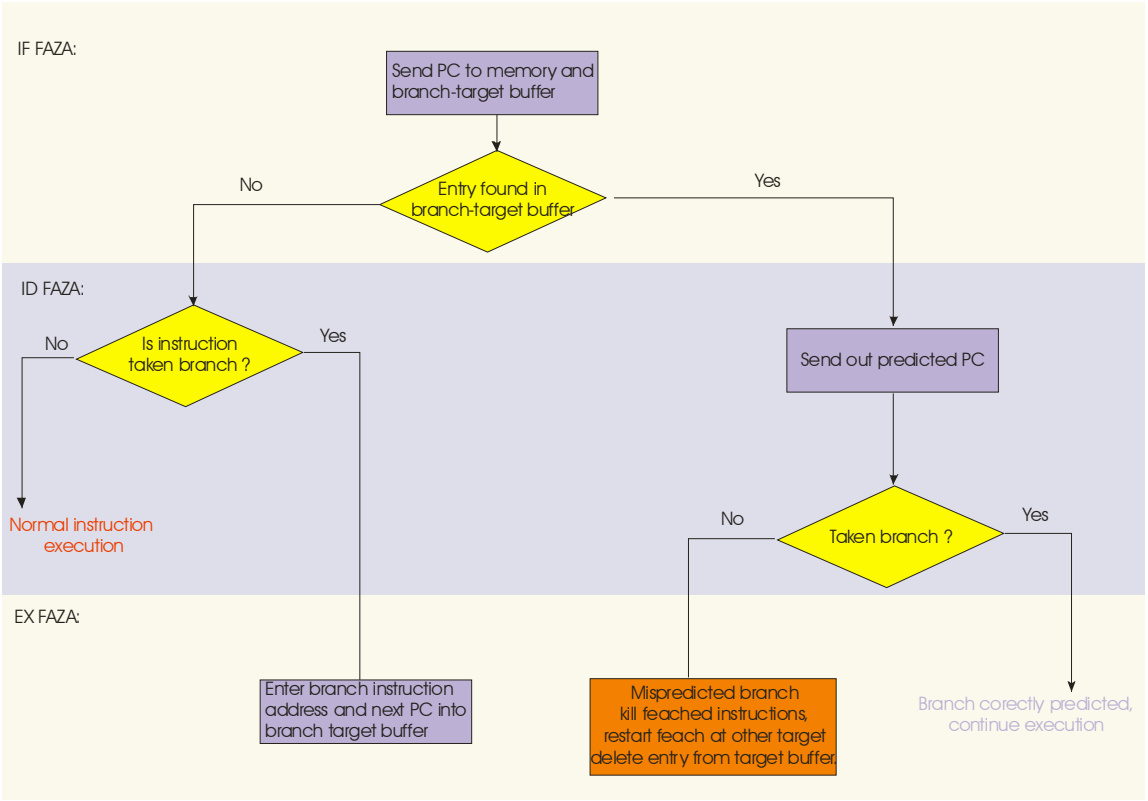
Broj izgubljenih ciklusa takta zavisi od toga kolika je pouzdanost predviđanja, kolika je cena kod promašaja ili pak kolika je cena ukoliko ne nađemo instrukciju u BTB–u.

Aktivnosti po fazama izvršenja instrukcije su:

U IF fazi se pristupa BTB–u na osnovu PC–a ispituje se da li je to već neka poznata instrukcija grananja. Ako jeste, onda se na kraju IF faze zna adresa sledeće instrukcije i počinje se sa pribavljanjem te sledeće instrukcije. Ako pak željena instrukcija ne postoji u BTB–u, onda se u ID fazi instrukcije ispituje da li je instrukcija koja je pribavljena grananje koje se obavlja. Ako nije, nastavlja se mormalno sa izvršavanjem programa (ne menja se sadržaj PC–a), a ako jeste mora da se modifikuje BTB. Adresa ove instrukcije se upisuje u BTB, kao i adresa grannja.

Ako je instrukcija pronađena u BTB–u ispituje se da li se grananje obavlja – ako se obavlja onda nemamo gubitke (zastoje), a ako se ne obavlja onda treba modifikovati BTB i pogrešno pribavljenu instrukciju ubiti (poništiti) i početi sa pribavljanjem prave instrukcije.

Sve ovo je slikovito predstavljeno na sledećoj slici.



Na ovoj slici mogu se jasno videti aktivnosti o fazama izvršenja instrukcije.

Ako se adresa instrukcije nađe u baferu, tada je pribavljena instrukcija, instrukcija grananja, koja je predviđena kao taken; tako pribavljanje nove instrukcije započinje sa predviđene adrese u ID fazi. Ako se adresa ne pronađe u BTB–u, a kasnije se ustanovi da je u pitanju grananje koje se obavlja, bafer se ažurira zajedno sa ciljnom adresom grananja, koja je poznata na kraju ID faze. Ako je adresa u BTB–u pronađena, ali se ispostavi da je u pitanju not taken instrukcija grananja, ona se uklanja iz bafera. Ako je bila u pitanju naredba grannja i ako je pronađena u BTB–u i ako je predviđanje bilo tažno, tada se izvršavanje nastavlja bez ikakvog zastoja.

Ako je predikcija nekorektna, gubi se 1 ciklus takta kod pribavljanja pogrešne insrukcije, pa se vrši pribavljanje korektne u sledećem ciklusu takta.

Sa slike se može videti da neće biti kašnjenja tj. gubitaka zbog naredbe grananja ako se u baferu pronađe naredba grananja i ako je predikcija tačna. u suprotnom javiće se gubitak od najmanje 2 ciklusa takta. U praksi, ovaj gubitak može biti veći jer se BTB mora ažurirati, a kapacitet mu je ograničen.

Da vidimo kolika je efikasnost ove tehnike.

Primer: Odrediti srednje kašnjenje za BTB usvajajući da su kašnjenja u navedenoj tabeli. Usvojiti da se pouzdanost predviđanja 90%, da je stopa pogodaka u BTB–u takođe 90% i da se grananje izvršava u 60% slučajeva.

Instrukcija u baferu	Predikcija	Akcija	Izgubljeno ciklusa
DA	T	T	0
DA	T	NT	2
NE	/	T	2

Srednje kašnjenje =SK

Stopa pogodaka u BTB–u =SPogBTB

Procenat pogrešnih predikcija =PPP

Stopa promašaja u BTB–u =SPromBTB

Procenat obavljenih grananja =POG

Ako uvedemo ove oznake možemo da napišemo formulu:

SK = (SPogBTB * PPP*2) + (SPromBTB * POG * 2)

=0.9 * 0.1 * 2 + (1–0.9) * 0.6 * 2 = 0.18+0.12 = 0.3 CT

To znači da su gubici zbog naredbi grananja u proseku svedeni na 0.3 ciklusa takta.

Jedna modifikacija BTB–a je da se zapamti ciljna instrukcija umesto, ili pored, predviđene ciljne adrese. Ova modifikacija ima jednu potencijalnu prednost. Naime, ovo dozvoljava da pristup BTB–u traje duže od vremena između dve uzastopne FETCH faze. Tj. duže od vremena trajanja FATCH faze.

Ovo može da dozvoli korišćenje većih BTB–a

Do sada smo se upoznali sa nekoliko softverskij (statičkih) i hardverskih (dinamičkih) tehnika za poboljšanje performansi protočnog sistema. Ove tehnike su napadale zavisnosti po podacima (Scoreboard, Tomasulov algoritam, odmotavanje petlje) i kontrolne hazarde (zakašnjeno grananje, BPB i BTB).

Do sada nam je osnovni zadatak bio postizanje protočnosti sistema od jedne instrukcije u jednom ciklusu takta.

Sada ćemo se pozabaviti tehnikama koje pokušavaju da eksploatišu veći ILP izdavanjem više instrukcija u jednom ciklusu takta.

Dalje smanjenje kašnjenja:

Tehnika odmotavanja petlje ima za cilj da poveća količinu raspoloživog ILP–a.

Scoreboard tehnika i Tomasulov algoritam su imale za cilj da se postigne idealni CPI=1 instrukcija/clock. Da podsetimo CPI je srednji broj taktova potreban da se jedna instrukcija izvrši.

BPB i BTB su imale kašnjenja zbog naredbi grananja. Sve ovo je bilo sa ciljem da se postigne idealni CPI.

CPI je može biti manji od 1 ako se pribavlja i izdaje jedna instrukcija u ciklusu takta. Da bi se CPI dalje redukovao potrebno je ostvariti pribavljanje i izdavanje više od jedne instrukcije po ciklusu takta. Rad superskalarnih računra se upravo bazira na ovoj ideji.

Superskalarni računari mogu da izdaju različit broj instrukcija po ciklusu takta. Kod tipičnog superskalarnog računara hardver može da izda između jedne i osam instrukcija po ciklusu takta (u zavisnosti od raspoloživog ILP–a).

Nasuprot Superskalarnim računarioma, VLIW izdaje fiksan broj instrukcija koje su formirane ili kao jedna velika instrukcioja ili kao fiksnipaket. VLIW su po definiciji sa statičkim planiranjem.

Superskalarni računari mogu da koriste statičko planiranje izvršenja (uz pomoć kompilatora) ili dinamičko koje se zasniva na Scoreboard tehnici ili Tomasulovom algoritmu.

Da bi smo objasnili ideju na kojoj se zasniva rad Superskalarnih procesora iskoristićemo istupetlju kao u prethodnim primerima.

Dakle, za dalje smanjenje broja ciklusa takta kašnjenja po instrukciji, neophodno je da se u jednom ciklusu takta pribavlja više instrukcija, što je moguće kod superskalarnih procesora. Da bi oni mogli da se primene, količina ILP–a (instruction level paralelism) mora da bude zaista velika. Ukoliko je ILP malo onda ne možemo da pribavljamo dovoljan broj nezavisnih instrukcija. U ovom slučaju vršiće se i istovremeno izvršavanje instrukcija koje su istovremeno pribavljene. Kod prosečnog superskalarnog procesora vrši se pribavljanje 1–8 instrukcija istovremeno. Da bi instrukcije mogle da budu pribavljene,one moraju da budu nezavisne. Kompilator ili hardver (moguće su obe varijante) pronlazi ove instrukcije. Dakle, superskalarni procesori mogu biti sa statičkim ili dinamičkim preuređenjem instrukcija.

VLIW (Very Long Instruction Word):

U svakom Ciklusu takta se pribavlja jednak broj instrukcija (ne kao kod super skalarnog procesora gde broj ribavljenih instrukcija može da varira od 1 do 8), a te instrukcije se zadaju kao jedan instrukcioni paket fiksne veličine ili kao jedna veoma dugačka instrukcija. U obe varijante reč je o statičkom preuređenju instrukcija (to radi kompilator).

Usvojimo da imamo superskalarni procesor koji može maksimalno da pribavlja istovremenom 2 instrukcije. Posmatramo izvršenje petlje gde imamo sabiranje skalara sa vektorom:

X(i)=X(i)+S

Usvojimo iste latencije za funkcionalne jedinice kao kada smo počeli da govorimo o odmotavanju petlje:

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

Ovo je naravno, neodmotana petlja.

Instrukcije koje se pribavljaju zajedno treba da ne traže istovremeno pristup istom hardveru (kako ne bi došlo do strukturalnih hazarda). Ako usvojim da je prva instrukija koja se pribavlja neka Integer instrukcija (tu spadaju i LOAD i STORE i BRANCH), a druga neka FPALU instrukcija. Tada će šansa biti

mala da nastane strukturalni hazard, jer ovae naredbe ne koriste iste hardverske resurse (registre i iste funkcionalne jedinice). Pa zašto je onda šansa mala, a nije jednaka nulu ? Zato što one ipak koriste istu memoriju, pa šansa za nastajanje strukturalnog hazarda ipak postoji (naravno, ukoliko memorija nema dva porta za čitanje). Ovo se rešava uvođenjem ograničenja da jedna instrukcija nesme više od jedared da se obrati memoriji(u jednom ciklusu takta). Ako neka instrukcija u nizu instrukcija zavisi od prethodne ili ne zadovoljava kriterijum za izdavanje, samo instrukcije koje joj prethode biće izdate.

Da bi smo objasnili ideju na kojoj se zasniva rad Superskalarnih procesora, usvojimo da se dve instrukcije mogu izdavati po ciklusu takta. Jedna od instrukcija može biti LOAD, STORE, BRANCH ili INTEGER ALU operacija, a druga može biti FP operacija.

Uvek je prva instrukcija u paru Integer instrukcija. Druga instrukcija se može izdati samo ako se prva može izdati.

Izdavanje integer instrukcije paralelno sa FP operacijom je mnogo manjezahtevno od izdavanja dve proizvoljne instrukcije. Ova koncepcija je veoma slična organizaciji procesora HP 7100.

Sa ovakvim nažinom rada smo znčajno povećali brzinu izdavanja FP instrukcija. Da bi ovo moglo da funkcioniše neophodno je da FP funkcionalne jedinice budu potpuno protožne ili da ih ima više nezavisnih. U protivnom bi FP data path (staza podataka) brzo postala usko grlo.

Izdavanjem integer i FP instrukcija paralelno, javlja se potreba za dodatnim hardverom pored uobičajene detekcije hazarda. Ova potreba je minimizirana. Naime, integer i FP operacije koriste različite skupove registara i razližite funkcionalne jedinice od LOAD/STORE instrukcija.

Jedine poteškoće nastupaju kada je integer instrukcija FP LOAD, FP STORE tj. kada koristi FP registre. Ovaj hazard se može eliminisati sa dva dodatna porta za čitanje i upis na FP registarskom fajlu.

Druga poteškoća koja može ograničiti efikasnost superskalarnih raunara je to što je latentnost LOAD instrukcije 1 ciklus takta, pa se kod superskalarnih procesora rezultat LOAD ne sme koristiti u istom i u narednom ciklusu takta. To znači da sledeće tri instrukcije ne mogu koristiti rezultat LOAD naredbe bez zaustavljanja. Kašnjenje za BRANCH, takođe može biti tri instrukcije, jer BRANCH mora biti prva u paru.

Da bi se efikasno iskoristio raspoloivi ILP, kod superskalarnih procesora, potrebne su moćni kompilatorske ili harverske tehnike za planiranje izvršenja instrukcije.

Da vidimo sada kako se odmotavanjem petlje sa preuređianjem instrukcija može lepo iskoristiti kod superskalarnih procesora.

Da bi se izbeglo zaustavljanje protočnog sistema potrebno je da postoji pet kopija tela petlje.

Dakle, odmotana petlja, bez preuređenja instrukcija izgledala bi ovako:

Ako su latencije:

Instrukcija koja generise rezultat	Instrukcija koja koristi rezultat	Latencija
FPALU	FPALU	3
FPALU	SD	2
LD	FPALU	1
LD	SD	0

Sa latencijama datim u tabeli, da vidimo kolko je trajanje jedne instrukcije.

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

FOR i:=1 to 1000 do
X(i)=X(i)+S

LOOP:	LD	F0,0(R1)
	ADDD	F4,F0,F2
	SD	(R1),F4
	LD	F6,–8(R1)
	ADDD	F8,F6,F2
	SD	–8(R1),F8
	LD	F10,–16(R1)
	ADDD	F12,F10,F2

	SD	–16(R1),F12
	LD	F14,–24(R1)
	ADDD	F16,F14,F2
	SD	–24(R1),R16
	LD	F18,–32(R1)
	ADDD	F20,F18,F2
	SD	–32(R1),F20
	SUBI	R1,R1,#40
	BNEZ	R1,LOOP

Ovako bi izgledala odmotana petlja 5 puta, ali ne preuređena. Kada bi je preuredili, dobili bi petlju prikazanu neuređene petlje.

Integer:		Floating Point (FP)
Loop:	LD F0,0(R1)	
	LD F6,–8(R1)	
	LD F10,–16(R1)	ADDD F4,F0,F2
	LD F14,–24(R1)	ADDD F8,F6,F2
	LD F18,–32(R1)	ADDD F12,F10,F2
	SD 0(R1),F4	ADDD F16,F14,F2
	SD –8(R1),F8	ADDD F20,F18,F2

	SD –16(R1),F12	
	SD –24(R1),F16	
	SUBI R1,R1,#40	
	BNEZ R1,Loop	

U ovom slučaju imamo 12 Ciklusa takta po 5 instrukcija, što je još bolje nego za odmotanu i preuređenu petlju.

2.4 ciklusa takta / elementu naspram 3.5 ciklusa takta po elementu kod standardnog protočnog procesora sa odmotavanjem i preuređenjem petlje.

Ako je pogodniji odnos između Integer i FP instrukcija onda može biti još bolje.

U ovom primeru performanse Superskalarnog procesora su ograničene onosom (balansom) između integer i FP izračunvanja. Svaka FP operacija se izdaje sa integer instrukcijom, ali nema dovoljno integer instrukcija da bi sve FP operacije imale svoj integer par. (da bi FP pipeline bio pun).

Iza load naredbe 3 naredbe ne smeju da zavise od LOAD (latencija je 1, ali sa load se pribavlja još jedna, i u tom sledećem ciklusu takta latencija je 2. Zanči 1+2=3). U proramu mora da postoji dovoljna količina ILP–a kako bi protočni sistem bio stalno pun.

Ovu petlju je neophodno odmotati najmanje 5 puta.

Šta je to Softverska protočnost i simboličko odmotavanje petlje?

Ovo je još jedna metoda koja omogućava povećanje raspoloživog ILP–a (bez povećanje dužine programskog koda. Kod tehnike odmotavanja petlje količina raspoloživog ILP–a se povećava tako što se kreiraju duže sekvence pravolinijskog koda, a zatim se vrši preuređenje instrukcija.

Kod softverske protočnosti, primenjuje se reorganizacija izvorne petlje tako da se svaka iteracija u novoj petlji (novom kodu) sastoji od instrukcija iz različitih iteracija izvorne petlje. Na taj način se razdvajaju međusobno zavisne instrukcije.

Softverska protožnost se ostvaruje preuređenjem različitih iteracija a bez odmotavanja petlje.

Ova tehnika je softvwerski pandan Tomasulovom algoritmu.

Uzmimo istu polaznu petlju, kao i u prethodnim primerima.

Softverski protočna–preuređena petlja će sadržati jednu LOAD, jednu ADDD i jednu STORE iz različitih iteracija. Pored toga postoji neki Start–up kod na početku i clean–up kod na kraju, koji vrše prilagođenje tj. korekciju koda. Postoji korekcija koda na početku i na kraju.

Ovom tehnikom se simbolički odmotava petljam, a zatim selektuje instrukcija iz svake iteracije. Pošto je odmotavanje simboližno, SUBI i BNEZ (loop overhead) instrukcije ne treba da se koriguju.

Evo kako izgleda telo odmotane petlje sa označenim insrukcijama koje se uzimaju iz svake iteracije.

Iteracija i	LD	F0,0(R1)
	ADDD	F4,F0,F2
	SD	0(R1),F4
Iteraija i+1	LD	F0,0(R1)
	ADDD	F4,F0,F2
	SD	0(R1),F4
Iteacija i+2	LD	F0,0(R1)
	ADDD	F4,F0,F2
	SD	0(R1),F4

Replicira se telo petlje bez SUBI i BNEZ tj. bez instrukcija za kontrolu petlje. A zatim se iz svake iteracije uzima po jedna instrukcija, i one uz dodatal LOOP CONTROLL instrukcija čine telo nove (softverski protožne petlje).

Selektovane insrukcije se zatim objedinjuju u petlju uz dodatak instrukcija za kontrolu petlje (loop controll). Te instrukcije su BNEZ i SUBI.

Pošto je odmotavanje simboličko, nema preimenovanja registara.

Sve ovo je lepo prikazano na slici ispod teksta.

	LD	F0,0(R1)	Start up kod
	ADDD	F4.F0,F2	Start up kod
	LD	F0,–8(R1)	Start up kod
LOOP:	SD	0(R1),F4	Pamti u M(i)
	ADDD	F4,F0,F2	ADD to M(i–1)
	LD	F0,–16(R1)	Loads to M(i–2)
	SUBI	R1,R1,#8	
	BNEZ	R1,Loop	
	SD	8(R1),F4	Clean up kod
	ADDD	F4,F0,F2	Clean up kod
	SD	0(R1),F4	Clean up kod

Start–up kod sadrži instrukcije iz prve idruge iteracije koje se neće izvršiti u petlji, a Clean–up kod instrukcije iz poslednje dve iteracije koje se neće izvršiti u petlji.

Instrukcije u telu petlje su iz tri različite iteracije i potpuno su nezavisne i mogu se izvršavati paralelno.

Prednost softverske protočnosti nad odmotavanjem petlje je kraći programski kod.

ZADACI (IZVRŠAVANJE PROGRAMA)

Zadatak br. 1

Za procesor koji ima RISC arhitekturu grafički predstaviti način serijskog izvršenja programa. Pri čemu su:

A,B i C registri

M memorijska lokacija

X,Y labele

Grafički predstaviti izvršavanje istog programa u slučajevima da procesor ima dvostepenu i trostepenu protočnu organizaciju, i naći ubrzanja koja se pri tome postižu ako se za otklanjanje diskontinuiteta pri izvršavanju programa (koga unosi naredba skoka) rešava pomoću:

- Tehnike zakašnjenog grananja
- Tehnike optimizovanog zakašnjenog grananja.

Rešenje zadatka br. 1

Prikazaćemo prvo kako bi izgledalo izvršavanje programa na sistemu bez protočne organizacije.

LOAD A,M	I	E	D																					
JZ A,X				I	E																			
LOAD C,M						I	E	D																
LOAD B,M									I	E	D													
ADD B,B,4												I	E											
JZ B,Y														I	E									
INC A																I	E							
Y:ADD A,B,3																		I	E					
JUMP X																				I	E			
INC B																								
X:SUB C,C,B																						I	E	
INC C																							I	E

Ovako izvršen program, potrošiće 25 ciklusa takta.

Ukoliko pak u sistemu postoji dvostepena protočnost i pritom se koristi tehnika zakašnjenog grananja, važiće sledeća tabela:

Dvostepena protočnost omogućava da u jednom trenutku bude najviše dve operacije u nekoj svojoj fazi. Uostalom, ovo potvrđuje i sama tabela.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LOAD A,M	I	E	D														
JZ A,X		I		E													
NOP				I	E												
LOAD C,M				I	E	D											
LOAD B,M					I		E	D									
ADD B,B,4							I		E								
JZ B,Y									I	E							
NOP										I	E						
INC A											I	E					
Y:ADD A,B,3												I	E				
JUMP X													I	E			
NOP														I	E		
INC B																	
X:SUB C,C,B															I	E	

INC C															I	E

Prodiskutovaćemo sve trenutke u izvršavanju programa:

01 trenutak: LOAD naredba ulazi u I fazu.

02 trenutak: JZ naredba ulazi u fazu I zato što je sistem dvostepen i u sistemu ima mesta

03 trenutak: LOAD naredba može da uđe u fazu D, jer je ništa u tome ne sprečava, ali naredba JZ ne može da uđe u svoju E fazu zbog toga što joj je potreban operand A iz prve naredbe (LOAD). Znači, ona mora da sačeka da naredba LOAD upiše svoj rezultat.

04 trenutak: Sada JZ naredba komotno može da uđe u svoju E fazu, naravno, može se izdati i naredba NOP u tom trenutku, a pošto je NOP naredba virtuelna naredba, zajedno sa njom se može izdati i sledeća LOAD naredba. Zašto se izdaje NOP naredba? Zato što dok se ne razreši naredba grananja, mi neznamo koja će naredba biti sledeća. Ovaj potencijal se može iskoristiti da se ubrza izvršenje, ali o tome ćemo u narednom slučaju govoriti tj. kada se govori o optimizovanom zakašnjenom grananju.

05 trenutak: U ovom trenutku, se najpre izvršava virtuelna nardba NOP, malopre izdata LOAD naredba može da uđe u svoju E fazu. U ovom trenutku se takođe izdaje i sledeća LOAD naredba.

06 trenutak: U šestom ciklusu takta, naredba LOAD C,M može da se završi, tj. uđe u D fazu. Malopre izdata LOAD naredba (LOAD B,M) ostaje zakočena, jer ne može da uđe u E fazu dok god naredba pre nje ne prestane da koristi memoriju (memorija ima jedan port)

07 trenutak: Sada je taj port oslobođen i naredba može da nastavi svoj životni ciklus. U to vreme izdaje se i naredna naredba.

08 trenutak: naredba LOAD B,M ulazi u svoju D fazu, a naredba posle nje ostaje zakočena, zato što joj je potreban operand B, koji se tek treba osvežiti.

09 trenutak: sada je taj operand osvežen i naredba nastavlja izvršenje, u isto vreme izdaje se naredna naredba.

Ovom logikom treba nastaviti do kraja programa!

Ukoliko pak u sistemu postoji dvostepena protočnost i pritom se koristi tehnika optimizovanog zakašnjenog grananja, važiće sledeća tabela:

LOAD A,M	I	E	D													
JZ A,X		I		E												
NOP				I	E											
LOAD B,M					I	E	D									
ADD B,B,4						I		E								
JZ B,Y								I	E							
LAOD C,M									I	E	D					

INC A										I		E				
Y:JUMP X												I	E			
ADD A,B,3													I	E		
INC B																
SUB C,C,B														I	E	
INC C															I	E

Jedina razlika u odnosu na prošli primer je ta što su naredbe u programu preuređene tako da ima manje zastoja. Ovakvim izvršavanjem programa potroši se 16 ciklusa takta. Dakle, ubrzanje u odnusu na slučaj kada nema protočnosti je 25/16.

Preuređenje programa se vrši tako, da se u slotove zakašnjenog grananja ubacuju umesto NOP naredbi neke naredbe koje se nezavisno od ishoda grananja izvršavaju.

Evo kako se vrši optimizacija.

1	LOAD	A,M
2	JZ	A,X
3	NOP	
4	LOAD	C,M
5	LOAD	B,M
6	ADD	B,B,4
7	JZ	B,Y
8	NOP	
9	INC	A
10	Y:ADD	A,B,3
11	JUMP	X
12	NOP	
13	INC	B
14	X:SUB	C,C,B
15	INC	C

Prva naredba ostaje ista (naravno). Druga naredba takođe ostaje ista. Pokušaćemo sada da naredbu NOP zamenimo nekom drugom, ali vidimo da se u slučaju grananja granamo na sam kraj pograma, te ne možemo da iskoristimo ovu NOP naredbu. Dakle, nepostoji naredba koja bi se izvršila nezavisno od toga da li se grananje obavilo ili ne.

Primećujemo sada da se C registar učitava u četvrtoj naredbi, a da se C kao operand koristi tek u 14–toj naredbi. Ovo možemo iskoristiti!!!!!

Iskoristićemo ga tako da umesto NOP operacije u liniji 8 stavimo naredbu koja učitava u C registar (jer se ionako pre toga C ne bi koristilo). Dakle, ovim smo eliminisali jedan NOP.

Takođe, može se primetiti da se registar A osvežava u naredbi 10 a da se nikada posle toga ne koristi. Dakle, umesto NOP naredbe u liniji 12 stavićemo naredbu koja osvežava registar A (linija 10).

Ovako optimizovan program iskoristio je dva slota zakašnjenog grananja.

1	LOAD	A,M
2	JZ	A,X
3	NOP	
4	LOAD	B,M
5	ADD	B,B,4
6	JZ	B,Y
7	LAOD	C,M
8	INC	A
9	Y:JUMP	X
10	ADD	A,B,3
11	INC	B
12	SUB	C,C,B
13	INC	C

Ukoliko pak u sistemu postoji trostepena protočnost i pritom se koristi tehnika zakašnjenog grananja, važiće sledeća tabela:

Trostepena protočnost omogućava da u jednom trenutku bude najviše tri operacije u nekoj svojoj fazi.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LOAD A,M	I	E	D														
NOP		I	E														
JZ A,X			I	E													
NOP				I	E												
LOAD C,M					I	E	D										
LOAD B,M						I	E	D									
NOP							I	E									
ADD B,B,4								I	E								
JZ B,Y									I	E							
NOP										I	E						
INC A											I	E					
Y:ADD A,B,3												I	E				
JUMP X													I	E			
NOP														I	E		
INC B																	
X:SUB C,B															I	E	
INC C																I	E

Uostalom, ovo potvrđuje i sama tabela.

Ukoliko pak u sistemu postoji trostepena protočnost i pritom se koristi tehnika optimizovanog zakašnjenog grananja, važiće sledeća tabela:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LOAD A,M	I	E	D												
NOP		I	E												
JZ A,X			I	E											
NOP				I	E										
LOAD B,M					I	E	D								
NOP						I	E								
ADD B,B,4							I	E							
JZ B,Y								I	E						
LOAD C,M									I	E	D				
INC A										I	E				
Y:JUMP X											I	E			
ADD A,B,3												I	E		
INC B															
X:SUB C,C,B													I	E	
INC C														I	E

Jedina razlika u odnosu na prošli primer je ta što su naredbe u programu preuređene tako da ima manje zastoja. Ovakvim izvršavanjem programa potroši se 15 ciklusa takta. Dakle, ubrzanje u odnusu na slučaj kada nema protočnosti je 25/15.

Preuređenje programa se vrši tako, da se u slotove zakašnjenog grananja ubacuju umesto NOP naredbi neke naredbe koje se nezavisno od ishoda grananja izvršavaju.

Zadatak br. 2

Za procesor koji ima RISC arhitekturu grafički predstaviti način serijskog izvršenja programa. Pri čemu su:

A,B i C registri

M memorijska lokacija

X,Y labele

Grafički predstaviti izvršavanje istog programa u slučajevima da procesor ima dvostepenu i trostepenu protočnu organizaciju, i naći ubrzanja koja se pri tome postižu ako se za otklanjanje diskontinuiteta pri izvršavanju programa (koga unosi naredba skoka) rešava pomoću:

- Tehnike zakašnjenog grananja
- Tehnike optimizovanog zakašnjenog grananja.

Rešenje zadatka br. 2

Prvo ću prikazati tok izvršavanja programa kada sistem ne bi bio protočan.

LOAD A,M	I	E	D																								
ADD A,A,5				I	E																						
JZ A,X						I	E																				
LOAD C,M								I	E	D																	
LOAD B,M											I	E	D														
SUB B,A,B														I	E												
JZ B,Y																I	E										
INC A																		I	E								
Y:ADD A,B,B																				I	E						
JUMP X																						I	E				
SUB C,C,B																											
INC B																											
X:SUB C,C,B																								I	E		
INC C																										I	E

Ovakvo izvršavanje programa bi trajalo 27 ciklusa takta. Sledeće tabele će pokazati kako se to vreme može smanjiti.

Ukoliko se isti program izvršavamo na sistemu sa **dvostepenom** protočnošću, ona ćemo imati sledeću situaciju:

LOAD A,M	I	E	D																
ADD A,A,5		I		E															
JZ A,X				I	E														
NOP					I	E													
LOAD C,M						I	E	D											
LOAD B,M							I		E	D									
SUB B,A,B									I		E								
JZ B,Y											I	E							
NOP												I	E						
INC A													I	E					
Y:ADD A,B,B														I	E				
JUMP X															I	E			
NOP																I	E		
SUB C,C,B																			
INC B																			
X:SUB C,C,B																	I	E	
INC C																		I	E

Uveli smo NOP operacije iza svake naredbe grananja.

Ovako izvršen program troši 19 ciklusa takta. Ubrzanje je dakle 27/19.

Ukoliko se isti program izvršavamo na sistemu sa **dvostepenom** protočnošću, i pri tome se koristi optimizacija, onda ćemo imati sledeću situaciju. Optimizacija se vrši tako što se umesto neke od NOP naredbi stavi naredba koja se izvesno izvršava bez obzira na ishod grananja.

Dakle,

LOAD A,M	I	E	D														
ADD A,A,5		I		E													
JZ A,X				I	E												
NOP					I	E											
LOAD B,M						I	E	D									
SUB B,A,B							I		E								
JZ B,Y									I	E							
LOAD C,M										I	E	D					
INC A											I		E				
Y:JUMP X													I	E			
ADD A,B,B														I	E		
SUB C,C,B																	
INC B																	
X:SUB C,C,B															I	E	
INC C																I	E

OVO ŠTO JE OZNAČENO CRVENIM NISAM SIGURAN. ALI MISLIM DA JE OVO KOČENJE JAVLJA ZBOG TOGA ŠTO D FAZA KORISTI REGISTAR FILE, PA SE NE MOŽE PRISTUPITI OPERANDU A.

Ovakvo izvršavanje je potrošili 17 ciklusa takta umesto 19 (kada je bilo neoptimizovano).

Što se tiče tokova podataka pri trostepenoj protočnosti, to nećemo readiti tj. neka to bude za vežbu.

Teorija vezana za statičko planiranje (odmotavanje petlje)

U svrhu povećanja paralelizma na nivou instrukcija se koriste različite metode. Da bi pipeline radio sa što manje zastoja paralelizam na nivou instrukcija je iskorišćen na taj način što se pronalaze sekvence nezavisnih instrukcija koje se mogu poklopiti u pipeline–u. Da bi se izbegli hazardi, instrukcija koja je zavisna od neke druge (predhodne) instrukcije mora biti razdvojena sa onoliko clock ciklusa kolika je latencija instrukcije od koje zavisi.

Mogućnost kompajlera da izvede ovakvo planiranje instrukcija naziva se statičko planiranje programa i izvodi se odmotavanjem petlje. Mogućnost kompajlera da izvede ovakvo planiranje zavisi i od već postojećeg paralelizma na nivou instrukcija u programu, kao i od latencija funkcionalnih jedinica u pipeline–u.

Zadatak br. 3

Na protočnom računaru izvršava se petlja:

FOR i=1 t 1000 DO

X(i) = x(i) + 5

a) Prevesti ovu petlju na asembler

b) Prikazati izgled petlje kada se uzimaju u obzir kašnjenja operacija za slučajeve kada se ne vrši i za slučaj kada se vrši planiranje redosleda izvršenja operacija.

Kašnjenja operacija u protočnom sistemu (pipe–line) su:

Instrukcija koja Generiše rezultat	Instrukcija koja koristi rezultat	Latencija
FP ALU	FP ALU	3
FP ALU	STORE FP	2
LOAD FP	FP ALU	1
LOAD FP	STORE FP	0

Naredba grnanja unosi kašnjenje od 1 ciklusa takta.

c) prikazati izgled razmotane petlje kada se u telo petlje umetaju četri kopije tela polazne petlje (uz pretpostavku da je ukupan broj izvršenja tela polazne petlje deljiv sa 4) u slučajevima kada se vrši planiranje i kada se ne vrši planiranje.

Rešenje zadatka br. 3

Prvo treba da prevedemo petlju na asembler.

LOOP	LD	F_0,0(R_1)
	ADDD	F_4,F_0,F_2
	SD	0(R_1),F_4
	SUBI	R_1,R_1,#8
	BNEZ	R_1,LOOP

Na početku asemblerskog programa se u registar F_0 učitava prvi element niza, koji se nalazi na lokaciji 0(R_1).

Zatim se u regstar F_4 smešta zbir tog elementa i neke konstante koja se nalazi u registru F_2.

Zatim se ovako osvežen element niza ponovo vraća na mesto odakle je i pročitan tj na lokaciju 0(R_1)

Posle ovoga, moramo da dekrementiramo brojač petlje, nebi li ovu operaciju izvršili tačno određen broj puta tj. onoliko puta koliki je R_1.

Na kraju programa ispitujemo uslov za grananje.

Sada ćemo napisati isti program, ali uvažavajući latencije (nećemo vršiti planiranje)

LOOP	LD	F_0,0(R_1)
	ADDD	F_4,F_0,F_2
	SD	0(R_1),F_4
	SUBI	R_1,R_1,#8
	BNEZ	R_1,LOOP

Ove zastoje ubacujemo na osnovu tabele. Tj. iz tabele se vidi, da ukoliko postoji zavisnost između LOAD i ALU naredbe onda treba uneti 1 ciklus zastoja.

Takođe, ukoliko postoji zavisnost između ALU naredbe i STORE naredbe, onda treba uvesti 2 ciklusa zastoja.

Na kraju, znamo da naredba grananja unosi 1 ciklus zastoja.

Ovakvo izvršavanje program bi trajalo 9 ciklusa takta, zato što su sada respektovane latencije među instrukcijama.

Malopre nismo ni pokušali da na neki način optimizujemo program, već smo samo dodali pauze tamo gde je trebalo. Ova situacija se može poboljšati tako što promenimo redosled izvršenja instrukcija ali tako da ne promeni SVRHA PROGRAMA tj. da se ne promeni sam program.

LOOP	LD	$F_0,0(R_1)$
	ADDD	F_4,F_0,F_2
	SUBI	$R_1,R_1,\#8$
	BNEZ	R_1,LOOP
	SD	$8(R_1),F_4$

Ovaj program će trajati 6 ciklusa takta, što je 3 ciklusa manje od predhodnog rešenja. Očigledno smo postični znatnu uštedu.

Ali kako smo je postigli?

Evo kako, odmah posle sabiranja rezultata u F_4 dekrementirali smo R_1 (koji je ustvari brojač petlje) i ispitali kraj petlje. Ovde može doći do zabune jer se može pomisliti da se poslednja (SD) naredba nikada ne izvršava tj. nikada se osveženi elementi ne vraćaju na svoje mesto. Međutim, to nije tačno jer (ukoliko niste zaboravili) naredba grananja ima latenciju od jednog klok ciklusa te će se uvek izvršavati i jedna instrukcija posle naredbe grananja (takozvani slot kašnjenja).

Moramo primetiti da je ova poslednja naredba modifikovana tako da umesto SD $0(R_1),F_4$ imamo SD $8(R_1),F_4$. Dakle, pošto je R_1 dekrementirano za 8, a mi treba da upišemo na adresu $0+R_1$, onda ćemo umesto 0 pisati 8. Tako da dobijamo $8+(R_1-8)$ što je ustvari isto zar ne? :)

c) Odmotavanje petlje

LOOP	LD	$F_0,0(R_1)$
	ADDD	F_4,F_0,F_2
	SD	$0(R_1),F_4$
	LD	$F_6,-8(R_1)$

	ADDD	F_4,F_6,F_2
	SD	$-8(R_1),F_8$
	LD	$F_{10},-16(R_1)$
	ADDD	F_{11},F_0,F_2
	SD	$-16(R_1),F_{12}$
	LD	$F_{14},-24(R_1)$
	ADDD	F_{16},F_{14},F_2
	SD	$-24(R_1),F_{16}$
	SUBI	$R_1,R_1,\#32$
	BNEZ	R_1,LOOP

U neuređenoj petlji, treba razlikovati telo petlje, od onog dela petlje koji služi za utvrđivanje da li će se petlja idalje vrteti.

Prilikom razmotavanja petlje umnožavamo telo petlje. Sve ostalo ostaje isto:

Dakle, na početku ponovo (kao i malopre) u F_0 učitavamo prvi element niza. Poštujući latenciju instrukcije LOAD (kada je vezana za ALU instrukciju) umećemo jedan ciklus takta zastoja. Zatim u registru F_4 vršimo sabiranje tekućeg elementa i konstante koja se nalazi u registru F_2.

Opet poštujemo latenciju i umećemo dva ciklusa takta zastoja.

Sada vraćamo osveženi element nazad (tamo odakle smo ga pročitali).

Sada je najbitniji deo:

Pošto se inače u petlji različite iteracije razlikuju jedino po tome što imaju različitu vrednost brojača, narednu iteraciju (u okviru tekuće) možemo simulirati tako što ćemo dekrementirati brojač. Zatim, u neki drugi registar učitamo naredni element niza, isvežimo ga, i vraćamo tamo odakle smo ga uzeli.

Sve se ovo ponavlja onoliko puta koliko puta treba da razlijemo petlju!

Na kraju, dekrementiramo osnovni brojač, ali za 4 puta veći iznos (8*4=32) zato što je u okviru jedne iteracije ovako odmotane petlje izvršeno 4 osnovnih iteracija.

Poslednja naredba je naredba grananja u kojoj se ispituje da li je petlja došla do kraja ili ima još da se vrti.

Ovoj petlji će biti potrebno 27 ciklusa takta da se izvrši. Ali, zato što ona u sebi sadrži 4 petlje, onda je efektivni broj ciklusa po originalnoj petlji 27/4 = 6.8

Setimo se da je ne odmotana i nepreuređena petlja trajala 9 ciklusa! Znači i bez optimizacije ušteda je primetna.

Kada bi sada preuredili odmotanu petlj, dobili bi smo sledeći program:

LOOP	LD	F_0,0(R_1)
	LD	F_6,–8(R_1)
	LD	F_{10},–16(R_1)
	LD	F_{14},–24(R_1)
	ADDD	F_4,F_0,F_2
	ADDD	F_8,F_6,F_2
	ADDD	F_{12},F_{10},F_2
	ADDD	F_{16},F_{14},F_2
	SD	0(R_1),F_4
	SD	–8(R_1),F_8
	SD	–16(R_1),F_{12}
	SUBI	R_1,R_1,#32
	BNEZ	R_1,LOOP
	SD	8(R_1),F_{16}

Preuređenje odmotane petlje se zasniva na istim principima na kojima se zasnivalo i preuređenje neodmotane petlje.

Ova petlja potrošiće 14 ciklusa takta, ali s obzirom da se u jednoj iteraciji ove petjle izvršava 4 iteracije osnovne petlje onda je broj ciklusa takta potrebnih da se obbavi jedna osnovna petlja jednak 14/4 = 3.5 što je izuzetno vreme.

Zadatak br. 4

Dat je asemblerski program:

LOOP:	LD	F_0,0(R_1)
	MULTD	F_0,F_0,F_2
	LD	F_4,0(R_2)
	ADDD	F_0,F_0,F_4
	SD	0(R_2),F_0
	SUBI	R_1,R_1,#8
	SUBI	R_2,R_2,#8
	BNEZ	R_1,LOOP

Neka su latencije date sledećom tabelom:

Instrukcija koja Generiše rezultat	Instrukcija koja koristi rezultat	Latencija
FP ALU	FP ALU	4
FP ALU	STORE FP	1
LOAD FP	FP ALU	2
LOAD FP	STORE FP	0

I neka naredba grananja unosi jedan ciklus takta zastoja.

Odmotati petlju najmanji broj puta tako da se ona može preurediti tako da nema ni jednog ciklusa takta zastoja između naredbi.

Rešenje zadatka br. 4

Prvo što ćemo uraditi je da napišemo asemblerski program poštujući latencije koje su date tabelom.

LOOP:	LD	F_0,0(R_1)
	MULTD	F_0,F_0,F_2
	LD	F_4,0(R_2)
	ADDD	F_0,F_0,F_4
	SD	0(R_2),F_0

	SUBI	R_1,R_1,#8
	SUBI	R_2,R_2,#8
	BNEZ	R_1,LOOP

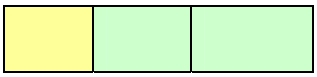
Nekoga može da zbuni to što se ovde javlja 3 ciklusa zastoja između LOAD i ADD instrukcije, a iz tabele vidimo da je nominalan broj zastoja 2?

Ovo se objašnjava time što se još uvek latencija prethodne naredbe (MULD) nije završila. Naime, ona traje 4 ciklusa takta, tako da nam se čini da ona dodaje jedan. Međutim, ona ne dodaje jedan ciklus takta već su svi ciklusi zastoja pod njenim uticajem. Ciklusi zastoja koji bi trebalo da potegnu od LOAD i ADDD naredbe su se uklopili u cikluse zastoja između MULTD i ADDD. Ovo je jako bitan zaključak !

OK.

Sada treba odmotati petlju dva puta. (prvo da probamo sa dva puta, pa ako neće, onda tri puta)

LOOP:	LD	F_0,0(R_1)
	MULTD	F_0,F_0,F_2
	LD	F_4,0(R_2)
	ADDD	F_0,F_0,F_4
	SD	0(R_2),F_0
	LD	F_6,–8(R_1)
	MULTD	F_6,F_6,F_2
	LD	F_8,–8(R_2)
	ADDD	F_6,F_6,F_8
	SD	–8(R_2),F_6
	SUBI	R_1,R_1,#16
	SUBI	R_2,R_2,#16
	BNEZ	R_1,LOOP



Uvodimo registar F_6 kako ne bi došlo do konflikta

Uvodimo registar F_8 kako ne bi došlo do konflikta

Ako pokušamo da ovu petlju preuredimo tako da nema zastoja, shvatićemo da je nemoguće. Dakle, petlju moramo odmotati tri puta.

Sledeći program sadrži tri puta razmotanu početnu petlju.

LOOP:	LD	F_0,0(R_1)	
	MULTD	F_0,F_0,F_2	
	LD	F_4,0(R_2)	
	ADDD	F_0,F_0,F_4	
	SD	0(R_2),F_0	
	LD	F_6,–8(R_1)	
	MULTD	F_6,F_6,F_2	
	LD	F_8,–8(R_2)	
	ADDD	F_6,F_6,F_8	
	SD	–8(R_2),F_6	
	LD	F_{10},–16(R_1)	

	MULTD	F_{10},F_{10},F_2	
	LD	$F_{12},-16(R_2)$	
	ADDD	F_{10},F_{10},F_{12}	
	SD	$-16(R_2),F_{10}$	
	SUBI	$R_1,R_1,\#24$	
	SUBI	$R_2,R_2,\#24$	
	BNEZ	$R_1,R_1,LOOP$	

Sada ovako razmotanu petlju treba preurediti tako da nema zastoja među naredbama:

LOOP:	LD	$F_0,0(R_1)$
	LD	$F_4,0(R_2)$
	LD	$F_6,-8(R_1)$
	MULTD	F_0,F_0,F_2
	LD	$F_8,-8(R_2)$
	MULTD	F_6,F_6,F_2
	LD	$F_{10},-16(R_1)$
	LD	$F_{12},-16(R_2)$
	MULTD	F_{10},F_{10},F_2
	ADDD	F_0,F_0,F_4
	ADDD	F_6,F_6,F_8

	SD	$0(R_2),F_0$
	SD	$-8(R_2),F_6$
	ADDD	F_{10},F_{10},F_{12}
	SUBI	$R_1,R_1,\#24$
	SD	$-16(R_2),F_{10}$
	BNEZ	$R_1,R_1,LOOP$
	SUBI	$R_2,R_2,\#24$

Učitava prvi element niza sa lokacije $0+R_1$ u registar F_0.

Sada bi trebalo da registar F_0 pomnožimo koeficijentom koji se nalazi u F_2, ali latencija koja se mora sačekati je 2 ciklusa takta. Tako da u sledeće dva takta možemo ubacitineke nezavisne naredbe zar ne?

Ove dve naredbe koje smo ubacili učitavaju: prvi element drugog niza sa lokacije $0+R_2$ i drugi element prvog niza sa lokacije $-8+R_1$.

Sada je prošla potrebna latencija i množenje može da se obavi!

Sada bi trebalo sabrati registar F_0 sa prvim elementom iz drugog niza (koji se nalazi u F_4). Međutim latencija koju treba sačekati je 4 ciklusa takta. Tako da možemo da ubacimo neke druge naredbe koje ne zavise od MULTD.

Mogli smo i da uvedemo odmah drugo množenje, ali taba bi se morao uvesti jedan ciklus zastoje jer između LD F_6 i MULTD treba da prođe 2 ciklusa takta. Zbog ovoga smo se odlučili da još jedared učitamo potrebne podatke tj. da učitamo drugi element drugog niza u registar F_8.

Nakon toga možemo izdati MULTD naredbu.

Dakle, popunili smo dve od četri naredbe koje treba da popune šupljinu između prve MULTD naredbe i ADDD naredbe. Treba da nađemo još dve nezavisne.

Učitavamo treći element prvog niza i treći element drugog niza.

Iako se ostvarila mogućnost da se sada izda i naredba ADDD i da se završi sa prvom iteracijom, treba misliti i o narednim MULTD naredbama, koje takođe imaju veliku latenciju u odnosu na svoju ADDD naredbu. Tako da sada izdajermo i posledenju MULTD naredbu.

Sada konačno možemo izvesti ono sabiranje.

Ako pogledamo malo bolje, videćemo da možemo odmah da izvedemo i drugo sabiranje.

Za treće sabiranje treba da prođu još dva ciklusa takta. Taman toliko nam treba da naredbom SD vratimo u memoriju sve što tereba da se vrati.

Sada izvršavamo poslednje sabiranje.

Da bi ovaj rezultat vratili u memoriju (naredbom SD) treba da protekne 1 ciklus takta. Njega možemo ispuniti regularnim dekrementiranje registra R_1.

Nakon toga upisujemo u memoriju.

Pošto treba iskoristiti i slot zakašnjenog grananja, koji će se javiti posle BNEZ naredbe, sada moramo izdati upravp tu naredbu.

I na kraju, dekrementiramo i registar R_2.

To, što je poslednja naredba posle naredbe grananja ne treba da vas čudi, jer se zbog kašnjenja naredbe grananja od jednog ciklusa takta ova naredba uvek izvršava.

Zadatak br. 5

Dat je asemblerski program:

LOOP:	LD	F_0,0(R_1)
	LD	F_4,0(R_2)
	MULTD	F_0,F_0,F_4
	ADDD	F_0,F_0,F_2
	SUBI	R_1,R_1,#8
	SUBI	R_2,R_2,#8
	BNEZ	R_1,LOOP

Neka su latencije date sledećom tabelom:

Instrukcija koja Generiše rezultat	Instrukcija koja koristi rezultat	Latencija
FP ALU	FP ALU	3
FP ALU	STORE FP	2
LOAD FP	FP ALU	1
LOAD FP	STORE FP	0

I neka naredba grananja unosi jedan ciklus takta zastoja.

Odmotati petlju najmanji broj puta tako da se ona može preurediti tako da nema ni jednog ciklusa takta zastoja između naredbi.

Rešenje zadatka br. 5

Prvo što ćemo uraditi je da napišemo asemblerski program poštujući latencije koje su date tabelom.

LOOP:	LD	F_0,0(R_1)
	LD	F_4,0(R_2)
	MULTD	F_0,F_0,F_4
	ADDD	F_0,F_0,F_2
	SUBI	R_1,R_1,#8
	SUBI	R_2,R_2,#8
	BNEZ	R_1,LOOP

Sada treba utvrditi koliko puta treba odmotati petlju da bi se otklonili svi zastoji. Po svemu sudeći to bi trebalo da bude 3 puta, ali ćemo mi probati šta bi bilo kada bi petlju odmotali dva puta.

LOOP	LD	$F_0,0(R_1)$
	LD	$F_4,0(R_2)$
	MULTD	F_0,F_0,F_4
	ADDD	F_0,F_0,F_2
	LD	$F_6,-8(R_1)$
	LD	$F_8,-8(R_2)$
	MULTD	F_6,F_6,F_8

	ADDD	F_6,F_6,F_2
	SUBI	$R_1,R_1,\#16$
	SUBI	$R_2,R_2,\#16$
	BNEZ	R_1,LOOP

Sada treba preraspodeliti naredbe tako da se izgube ovi ciklusi zastoja.

To možemo uraditi na sledeći način:

Ovo je moje rešenje i treba neko da mi ga proveri.

	LD	$F_0,0(R_1)$
LOOP	LD	$F_4,0(R_2)$
	LD	$F_6,-8(R_1)$
	LD	$F_8,-8(R_2)$
	MULTD	F_0,F_0,F_4
	MULTD	F_6,F_6,F_8
	ADDD	F_0,F_0,F_2
	ADDD	F_6,F_6,F_2
	SUBI	$R_1,R_1,\#16$
	SUBI	$R_2,R_2,\#16$
	BNEZ	R_1,LOOP
	LD	$F_0,0(R_1)$

Zadtak br. 6

Prikazati dinamičko planiranje izvršenja datog programa primenom:

- **Scoreboard tehnike (ako je na raspolaganju)**
  - 1 INTEGER funkcionalna jedinica (latencija=3 ciklusa takta)
  - 1 ADD funkcionalna jedinica (latencija=8 ciklusa takta)
  - 1 DIV funkcionalna jedinica (latencija=30 ciklusa takta)
  - i 2 MUL funkcionalna jedinica (latencija=20 ciklusa takta)

- **Tomasulovog algoritma (ako je na raspolaganju)**
  - 5 LOAD bafera (latencija=3 ciklusa takta)
  - 3 ADD rezervacione stanice (latencija=8 ciklusa takta)
  - 3 MUL rezervacione stanice (latencija=20 ciklusa takta)

LD	$F_1,23(R_1)$
ADDD	F_6,F_1,F_1
LD	$F_2,45(R_2)$
MULTD	F_6,F_2,F_1
LD	$F_3,30(R_3)$
SUBD	F_5,F_6,F_3
MULTD	F_5,F_1,F_6
DIVD	F_7,F_5,F_6
ADDD	F_6,F_2,F_3
MULTD	F_9,F_7,F_3
SUBD	F_7,F_5,F_3

Rešenje zadatka br. 6

Scoreboard tehnika:

LD	F_1,23(R_1)	1	2	5	6
ADDD	F_6,F_1,F_1	2	7	15	16
LD	F_2,45(R_2)	7	8	11	12
MULTD	F_6,F_2,F_1	17	18	38	39
LD	F_3,30(R_3)	18	19	22	23
SUBD	F_5,F_6,F_3	19	40	48	49
MULTD	F_5,F_1,F_6	50	51	71	72
DIVD	F_7,F_5,F_6	51	73	103	104
ADDD	F_4,F_2,F_3	52	53	61	74
MULTD	F_9,F_7,F_3	73	105	125	126
SUBD	F_7,F_5,F_3	105	106	114	115

U prvom trenutku se izdaje LOAD naredba.

U drugom trenutku LOAD naredba ulazi u fazu RO (read operands), ali se tada i izdaje ADDD naredba jer koristi drugu funkcionalnu jedinicu iako nema slobodne operande. Tj F_1 tek treba da se upiše.

U trećem trenutku ne može da se izda naredna LOAD naredba, zato što prva LOAD naredba još uvek čeka svoju latenciju (3clk) i time drži INT funkcionalnu jedinicu zauzetom. Što se ADDD naredbe tiče ona ne može da pribavi svoje operande jer je njen operand F_1, koji će biti upisan tek kada ga prva naredba upiše u svojoj (write result fazi).

U petom trenutku se završava prva LOAD naredba. ADDD naredba ne može da uzme svoje operande jer se F_1 još nije upisao. Naredna LOAD naredba ne može da se izda zato što je INT funkcionalna jedinica zauzeta.

U šestom trenutku prva LOAD instrukcija upisuje svoj rezultat. Ali i dalje se niša ne može više uraditi sa ostalim naredbama.

U sedmom trenutku ADDD uzima svoje operande a LOAD naredba se izdaje (sada je INT funkcionalna jedinica slobodna).

U osmom trenutku ADDD čeka svoju latenciju (još se izvršava). LOAD može da uzme svoje operande. MULTD ne sme da se izda zato što je njen destinacioni registar isti kao i destinacioni registar ADDD naredbe. Došlo bi do hazarda. Ovakve zavisnosti se ne propuštaju u fazu izdavanja.

U 11 trenutku se završava LOAD instrukcija. ADDD i dalje čeka svoju latenciju, a MULTD nesme da se izda dok ADDD ne upiše u registar F6. (zato što im je destinacioni registar isti).

U 12 trenutku LOAD naredba je upisala svoj rezultat i time se vanredosledno okončala. Ovo je dozvoljeno zato što time nije ugrozila predhodne naredbe.

U 15 trenutku se završava ADDD naredba, ali MULTD ne sme da se izda dok ADDD ne upiše svoj rezultat.

U 16 trenutku ADDD upisuje rezultat.

U 17 trenutku se konačno izdaje MULTD naredba.

U 18 trenutku MULTD može da uzme svoje operande zato što su slobodni. I izdaje se LOAD naredba.

U 19 trenutku LOAD uzima svoje operande zato što su dostupni a SUBD se izdaje. MULTD se izvršava.

U 22 trenutku LOAD je završila svoje izvršavanje i spremna je da upiše svoj rezultat.

U 23 trenutku LOAD može vanredosledno da upiše svj rezultat jer time ne utiče na nijednu predhodnu naredbu. SUBD nesme da pribavi svoje operande dok MULTD ne upiše svoj rezultat.

U 38 trenutku MULTD završava svoje izvršavanje i spremna je da u narednom trenutku upiše rezultat.

U 39 trenutku MULTD upisuje rezultat.

U 40 trenutku SUBD naredba može da uzme svoje operande jer su joj dostupni.

U 41 trenutku SUBD se još nije izvršila a MULTD se nesme izdati jer ima se destinacioni registru poklapaju.

U 48 trenutku se završila SUBD naredba, ali još uvek nije upisala rezultat.

U 49 Trenutku je SUBD upisala u registar

U 50 trenutku se konažno izdaje naredba MULTD

U 51 trenutku MULTD uzima operande i izdaje se DIVD

U 52 trenutku se ništa ne menja (Multd se i dalje izvršava a DIVD nesme da uzme operande dok MULTD ne završi izvršenje, ali se može izdati ADDD.

Daljeu analizu možete sami nastaviti.

Tomasulov Algoritam:

LD	F_1,23(R_1)	1	5	6
ADDD	F_6,F_1,F_1	2	14	15
LD	F_2,45(R_2)	3	7	8
MULTD	F_6,F_2,F_1	4	28	29
LD	F_3,30(R_3)	5	9	10
SUBD	F_5,F_6,F_3	6	37	38
MULTD	F_5,F_1,F_6	7	49	50
DIVD	F_7,F_5,F_6	8	80	81
ADDD	F_6,F_2,F_3	9	18	19
MULTD	F_9,F_7,F_3	30	101	102
SUBD	F_7,F_5,F_3	31	58	59

1: izdaje se prva LOAD naredba.

2: Izdaje se ADDD nareba zato što postoji slobodna rezervaciona stanica.

3: Izdaje se druga LOAD naredba jer postoji slobodan LOAD bafer.

4: Izdaje se MULTD naredba

5: Prošla je latencija prve LOAD naredbe (ona je izdata u prvom taktu, izršavala se u taktu 2,3 i 4 i u petom je gotova. Naravno, izdaje se još jedna LOAD instrukcija jer ima mesta.

6: Prva LOAD naredba upisuje u registar svoj rezultat i izdaje se SUBD naredba. Sada je LD upisala u registar F_1, koji je potreban ADDD naredbi. Dakle, ona svoje izvršenje počinje u ciklusu 6 i traje u ciklusima 6,7,8,9,10,11,12,13. Dakle, biće gotova tek u 14–om taktu.

Što se tiče druge LOAD naredbe, njeni operandi su odmah bili slobodni te je ona svoje izvršenj počela odmah u 4 ciklusu takta tj. završiće se u 7–om.

Što se tiče MULTD naredbe, ona da bi počela da se izvršava, mora da sačeka drugu load naredbu da upiše rezultat. Dakle, po svemu sudeći njeno izvršenje će početi tek u osmom ciklusu i trajaće do 27–og dakle, biće gotova u 28–om ciklusu takta.

Što se tiče LOAD treće naredbe njeno izvršenje će početi još u 6–om ciklusu i trajaće do 6+3–ćeg ciklusa a to znači da će biti završena u 9–om ciklusu.

Što se tiče SUBD naredbe ona mora da sačeka MULTD naredbu da bi joj upisala registar F_6. Dakle, njeno izvršavanje će početi u 29–om ciklusu jer se podatak pored u destinacioni registar upisuje i svuda gde je to potrebno. Dakle, ako počne u 29–om ciklusu SUBD će trajati do 36–og ciklusa. Dakle, biće gotova u 37–om ciklusu.

Izgleda da smo u ovom šestom koraku izanalizirali dosta narednih koraka zar ne? :–)

Dalje nastavite sami :)

Zadtak br. 7

Prikazati ydinamičko planiranje izvršenja datog programa primenom:

- **Scoreboard tehnike (ako je na raspolaganju)**
  - 1 INTEGER funkcionalna jedinica (latencija=2 ciklusa takta)
  - 1 ADD funkcionalna jedinica (latencija=5 ciklusa takta)
  - 1 DIV funkcionalna jedinica (latencija=30 ciklusa takta)
  - i 2 MUL funkcionalna jedinica (latencija=15 ciklusa takta)

- **Tomasulovog algoritma (ako je na raspolaganju)**
  - 5 LOAD bafera (latencija=2 ciklusa takta)
  - 2 ADD rezervacione stanice (latencija=5 ciklusa takta)
  - 3 MUL rezervacione stanice (latencija=15 ciklusa takta)

LD	$F_1,34(R_1)$
LD	$F_2,25(R_2)$
ADDD	F_8,F_1,F_2
LD	$F_3,30(R_3)$
SUBD	F_6,F_2,F_3
MULTD	F_8,F_3,F_6
DIVD	F_7,F_3,F_8
ADDD	F_8,F_1,F_2
MULTD	F_7,F_3,F_1
DIVD	F_5,F_7,F_1
ADDD	F_7,F_1,F_2

Rešenje zadatka br. 7

Scoreboard tehnika.

LD	F_1,34(R_1)	1	2	4	5
LD	F_2,25(R_2)	6	7	9	10
ADDD	F_8,F_1,F_2	7	11	16	17
LD	F_3,30(R_3)	11	12	14	15
SUBD	F_6,F_2,F_3	18	19	24	25
MULTD	F_8,F_3,F_6	19	26	41	42
DIVD	F_7,F_3,F_8	20	43	73	74
ADDD	F_8,F_1,F_2	43	44	49	50
MULTD	F_7,F_3,F_1	75	76	91	92
DIVD	F_5,F_7,F_1	76	93	123	124
ADDD	F_7,F_1,F_2	93	94	99	100

Prilikom ispitivanja da li se može izdati treba se postavljati sledeća pitanja:

- Da li je u tom trenutku već jedna naredba izdata
- Da li nije neka naredba preskočebna (nesme da bude)
- Da nije zauzeta funkcionalna jedinica
- Da destinacioni registar nije preklopljen

Pitanje koje se postravlja kada operacuija treba da upiše u registar:

- Da li je neka naredba koja je izdata još uvek u ISUE fazi tj. da li je pročitala operande.

Funkcionalna jedinica	Komada	Latencija
INT	1	2
ADD	1	5
DIV	1	30
MUL	2	15

Tomasulov algoritam:

LD	F_1,34(R_1)	1	4	5
LD	F_2,25(R_2)	2	5	6
ADDD	F_8,F_1,F_2	3	11	12
LD	F_3,30(R_3)	4	7	8
SUBD	F_6,F_2,F_3	5	13	14
MULTD	F_8,F_3,F_6	6	29	30
DIVD	F_7,F_3,F_8	7	60	61
ADDD	F_8,F_1,F_2	13	19	20
MULTD	F_7,F_3,F_1	14	30	31
DIVD	F_5,F_7,F_1	62	93	94
ADDD	F_7,F_1,F_2	63	69	70

Bafer/Stanica	Komada	Latencija
LOAD	5	2
ADDD	2	5
MULT	3	15

Vektorski računari

Da bi se povećala brzina izvršavanja protočnog procesora treba smanjiti periodu kloka, a za to treba projektovati dublji protočni sistem. Druga varijanta je da koristimo pribavljanje i izvršavanje više instrukcija **istovremeno**. Obe varijante dovode do hazarda po podacima. Što je dublja protočnost i što je veći broj instrukcija koje se istovremeno pribavljaju i izvršavaju veće su šanse za nastanak **hazarda po podacima**.

Kompleksnost logike za detekciju hazarda raste sa kvadratom broja instrukcija koje su istovremeno prisutne u protočnom sistemu. Zbog toga je projektovanje upravljačke jedinice dosta težak i obiman posao, pa se nameću neke gornje granice za dubinu protočnog sistma odnosno, za broj instrukcija koje se istovremeno mogu naći u protočnom sistemu (pribavljati i izvršavati zajedno).

Protočni istemi koriste keš memoriju da bi se rešio problem uskog grla CPU–Memorija, ali mnogi naučno–tehnički problemi imaju skupove podataka kojima se pristupa sa niskom lokalnošću, pa hijerarhijska organizacija memorije ovde ima loše performanse (jer je ona zasnovana na lokalnosti).

NPR: Matrica je smeštena po **kolonama**. Ako je aplikacija takva da zahteva pristup elementima kolone, onda bi keš memorija dobila dobre rezultate. Ali ako je potrebno pristupati redom elementima **vrste**, keš memorija neće dati dobre rezultate jer stalno imamo promašaje keša. Različitim bojama označene su različite **kolone**, primećujemo da je sadržaj ovakve memorije lokalizovan po kolonama, ali ukoliko bi pokušali da pristupamo po vrstama, javljali bi se promašaji (ne može da bude lokalizovano i po vrstama i po kolonama).

U ovakvom slučaju su performanse koje se dobijaju sa kešom lošije od performansi koje se dobijaju kada keša ne bi bilo. Zato je nekad memorijska hijerarhija loša. Sve ovo se uspešno rešava korišćenjem **vektorskog računara**.

Vektorski procesor je procesor koji je specijalno projektovan za izračunavanja nad linearnim poljima (vektorskim podacima).

NPR: Jedna vektorska instrukcija definiše ceo posao oko sabiranja elemenata 2 vektora (svi se sabiraju u isto vreme). Vektorski računari i same vektorske instrukcije imaju karakteristike koje čine da se malopre nabrojani problemi lako rešavaju ili čak izbegavaju.

Izračunavanje svakog elementa vektora je nezavisno od prethodnog izračunavanja tog vektora, što dozvoljava korišćenje dubokih protočnih sistema bez mogućnosti za javljanje hazarda podataka. Sama vektorska instrukcija garantuje da nema zavinosti među izračunavnjima za različite elemente vektora. Odsustvo hazarda je ispitano od strane kompilatora, koji je odlučio da tu može da koristi vektorske instrukcije, ili pak od strane programera korišćenjem nekog višeg programskog jezika.

Vektorska instrukcija obavlja ogroman posao. Ona je ekvivalntna izvršenju cele jedne petlje u kojoj se u svakoj iteraciji vrši izračunavanje jednog elmenta polja, zatim, ažuriranje indeksa da bi se pristupilo sledećem elemntu polja i grananje na početak petlje. Umesto cele jedne petlje koristi se samo jedna instrukcija. To je divno zar ne?

Primer:

Običan program		Paralelni program
For	I=1 to n do	
	A(i)=b(i)+c(i)	A(1:n)=B(1:n)+C(a:n)
End for		

Upadljivo je prostije i za programera i za mašinu da izvrši ovaj niz instrukcija paralelno, u odnosu na noramlno (sekvencijalno) izvršavanje.

Vektorske instrukcije je zgodno koristiti kada imamo vektorske podatke, ali je nepoželjno koristiti ih kada su podaci skalarni.

Najčešće je oblik pristupa podacima poznat (sukcesivbne memorijske lokacije, vrste matrice, kolone matrice, dijagonala matrice) tj. nije nasumični pristup. Adekvatnom organizacijom memorije i adekvatnim smeštanjem podataka u memoriju, vektorskim procesorima se mogu postići bolje performanse nego ako se koristi keš. Kontrolni hazardi koji bi nastali zbog grananja na početak petlje ovde ne postoje jer je petlja zamenjena jednom instrukcijom (vektorskom). Pokazalo se, da se dobrim vektorskim kodom, mogu postići i do 10–20 puta bolje performanse nego sekvencijalnim kodom.

Vektorske insrukcije se koriste kod aplikacija koje koriste polja i tu možemo koristiti značajno poboljšanje performansi (ovo se koriasti kod vremenske prognoze, simulacije vožnje, kod multimedijalnih aplikacija itd ...)

Protočnost se koristi i kod vektorskih računara i to ne samo kod izvršenja aritmetičko logičkih instrukcija, već i kod izražunavanja efektivene adrese i kod pristupa memoriji, kako bi se dobile bolje performanse (pošto se ne koristi keš memorija).

Vektorski procesori pod određenim uslovima mogu izvršavati i više vektorskih instrukcija koje ne koriste iste funkcionlne jedinice, čime se uvodi i **paralelizam** u obradi instrukcija.

Osnovne komponente vektorskih računara su:

- Memorijski podsistem
- Skalarni podsistem
- Vektorski podsistem

Memorijski podsistem:

Glavna memorija vektorskog računara je po funkciji ista kao i kod skalarnih računara. Ona sadrži podatke u vektorskom i skalarnom obiliku i narvno instrukcije. Međutim zbog većeg broja obraćanja memoriji ova memorija je znatno složenija i njena organizacija direkno utiče na performanse celog sistema.

Memorija vektorskog računara je organizovana kao niz memorijskih banaka (banki) kojima se može jednovremeno pristupiti. Za smeštanje podataka se koristi **low order interleaving**. Tj. metoda preklapanja po nižim botovima adrese.

Adresa podatka (u kojoj banci ?)	O kojem je modulu (banci) reč ?

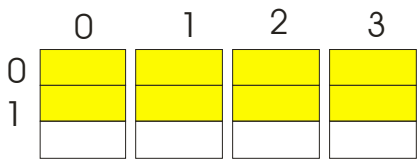
Adresa je uslovno podeljena na dva dela: Viši deo predstavlja adresu u okviru banke (modula) a niži deo predstavlja sam memorijski moduo (tj, njegovu adresu).

NPR:

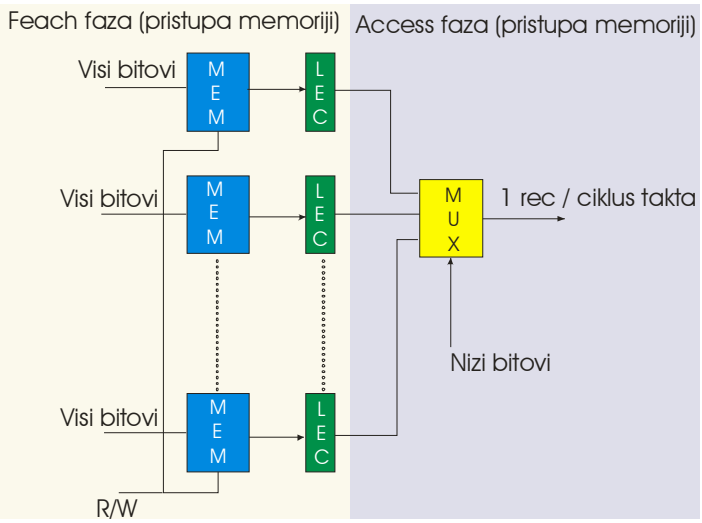
3bitne adrese, 4 memorijske banke.

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Poslednja kolona označava koja je banka u pitanju. Dakle, ovim smo postrigli da se memorijske lokacije koje se inače nalaze sukcesivno, jedna pored druge, sada nalaze u različitim bankama, što omogućava da se podacima pristupa paralelno (zato što se nalaze u različitim banakama, te je hardver koji se koristi različit, inače bi trebalo da se n puta pristupi istojmemorijskoj banci). Dakle, ostvaren je paralelni pristup sukcesivnim memorijskim adresama.



Podatak na adresi 001 se nalazi na adresi 0 ali modul 1. Ovakvim načinom smeštanja je postignut da se podaci koji se nalaze na sukcesivnim memorijskim adresama nalaze u susednim memorijskim bankama.



Za pristup podacima u banci koriste se **viši bitovi** adrese.

U odnosu na to odakle vektorske instrukcije pribavljaju operande, vektorski računari se dele na:

Vektorsko – registarske arhitekture

Memorijsko – memorijske arhitekture

Vektorsko – registarske (pandan RISC procesorima). Kod njih za obraćanje memoriji su LOAD i STORE a sve ostakle instrukcije pribavljaju podatke iz nekog od vektorskih ili skalarnih registara. Svi savremeni vektorski računari su ovakvi.

Memorijsko – memorijske arhitekture operandi su u memoriji (pandan CISC procesoru). Prvi vektorski računar Cyber 205 bio je ovog tipa ali su kasnije ovi računari prestali da se proizvode. Savremeni vektorski računari koriste do 1K memorijskih modula za smeštanje podataka. (vektorski računari su komercijalno počeli da se proizvode pre RISC procesora).

LVWS i **SVWS** se koriste za učitavanje elemenata koji nisu u sukcesivnim memorijskim lokacijama, a onda dobijemo logički susedne elemente.

LVI i **SVI** se koriste za retko posegnute strukture (za njihovo smeštanje se koristi vektor za pamćenje logičkog položaja tog elementa.

VLR je registar gde se pamti dužina vektora.

Skalarni podsistem: Je organizovan kao kod protočno organizovanog (standardnog) procesora. Samo ukoliko se naiđe na vektorski podatak može se preći na vektorski podsistem.

Vektorski podsistem: Ima registre za pamćenje operanada i protočno organizovane funkcionalne jedinice za izvršavanje operacija nad tim vektorskim podacima (registrima). **Memorijski pojas** se sastoji od 16 memorijskih banaka po 64K reči, brzina pribavljanja je oko 300 miliona reči u sekundi. **Skalarni pojas** postoji jer u programu uvek postoji deo koji ne može da se vektorizuje, dakle, za izračunavanje efektivne adrese instrukcije za pristup memorji skalarnog podsistema. On ima 8 registara opšte namene za smeštanje skalarnog podatka i 64 pomoćna registra za smeštanje (T00 do T77 –oktalno) za privrmeno pamćenje sadržaja S registara. Funkcionalne jedinice mogu uzimati podatak samo iz S registara.

Postoje i **skalarne funkcionalne jedinice** koje su potpuno protočno organizovane i tu imamo ADD, LOGICAL, SHIFT i POP17 (broji koliko jedinica ima u vektoru maske). Ove funkcionalne jedinice koja sam nabrojao su integer funkcionalne jedinice.

Za FP podatke se koriste **funkcionalne jeinice** koje su **zajedničke** za skalarne i vektorske podatke: ADD, MULT, RECIP (za recipročnu vrednost). U skalarni pojas spadaju i adresni registri A0–A7 i pomoćni adresni registri B0–B77 (oktalne vrednosti tj. ima ih 64).

A registri se koriste za adresiranje memorijskih banaka. Neki se mogu koristiti za pamćenje bazne adrese a neki za razmeštaj (pri izračunavanju efektivne adrese). Po toj adresi se u interni bafer smešta pribavljena instrukcija.

Vektorski podatak se sastoji od 8 vektorska registra V0–V7. služi za smeštanje vektorskih operanada i rezultata. Svaki vektorski registar je veličine 64 elemenata pri čemu je svaki element 64bita. Ako je vektor koji treba da se zapamti duži od 64 elemenata on se mora razbiti na delove po 64 elementa. Zato se koristi **VL** registar (dužine vektor).

Svaki vektorski registar ima dva porta za čitanje i jedan za upis. Postoje posebne vektorske funkcionalne jedinice za INTEGER, ADD, LOGICAL, SHIFT i sve zajedničke za FP koje su protočno orijentisane. Vektorske funkcionalne jedinice mogu pribavljati operande iz vektorskih registara ili iz skalarnih. Rezultati se takođe mogu smeštati u vektorske ali i u skalarne registre. Sve funkcionalne jedinice su protočno organizovane , sve operacije se izvršavaju za 1 takt.

VM (vektor maske) je 64 – bitni registar i svaki bit odgovara tačno jednom elementu vektorskog registra.

Postoje operacije koje se rade pod **maskom** (dakle, ne za sve elemente vektora, recimo da se izvbegne deljenje nulom).

Ukoliko se u VM nađe jedinica to znači da se nad tim elementom obavlja operacija a 0 da se ne obavlja. Noviji vektorski računari imaju paralelnu vektorsku Funkcionalnu jedinicu.

Paralelni računarski sistemi (Elektronski Fakultet – NIŠ)

Marko Miličić

Što se tiče formata instrukcije u CRAY 1 računaru su bile 32b LOAD i STORE, a sve ostale su bile 16–bitne.

Y (op. kod)	H (op. kod)	I (registar rezultata)	J (prvi operand)	K (drugi operand)
4b	3b	3b	3b	3b

Za binarbne operacije

Y (op. kod)	H (op. kod)	I (registar rezultata)	Shift ili maska
4b	3b	3b	3b 3b

Za unarne operacije

G (op. kod)	H (index reg)	I	JKN
4b	3b	3b	22b

LOAD i STORE

H(index) Neki od A registara.

I Registar operanda ili registar rezultata (neki V registar ili neki S registar u zavisnosti da li je vektor ili skalar u naredbi LOAD ili STORE.

JKN Specificira u slučaju skalara adresu jedne 64–bitne reči u glavnoj memoriji, a za vektor to je startna (bazna) adresa od koje počinje upis ili čitanje vektora a za dalje indeksiranje se koristi h tj. indeksni registar A.

PRIMER:

$y = a \cdot x + y$: x,y su vektori, a je skalar!

Usvojićemo da je broj elemenata vektora 64 tj. tačno kao veličina vektorskog registra. Prikazati kako izgleda skalarni i vektorski kod. Usvojiti da se startne adrese za x i y nalaze u Rx i Ry.

	LD	F0,A

	ADDI	R4,Rx,#512
LOOP:	LD	F2,0(Rx)
	MULTD	F2,F2,F0
	LD	F4,0(Ry)
	ADDD	F4,F4,F2
	SD	0(Ry),F4
	ADDI	Rx,Ry,#8
	ADDI	Ry,Ry,#8
	SUB	R20,R4,Ry
	BNEZ	R20,LOOP

Prvo se u registar F0 učitava skalar A.

Zatim se u registar R4 smesti zbir bazne adrese X niza i broja 512. Ovo se radi zato da bi se početna adresa X niza uvećala za 64*8 što je jednako broju bitova koji se nalaze u nizu X pod pretpostavkom da su podaci osmobitni a da se niz sastoji od 64 elemenata. Tj. ovo će se iskoristiti za ispitivanje kraja petlje.

Zatim se u F2 učita prvi element niza X, pomnoži se sa skalarom koji se nalazi u F0.

Zatim se učita prvi element niza Y i sabere se sa onim što smo dobili malopre tj. sa registrom F2.

Kada se sve ovo završi ovako osveženi element drugog niza se vraća tamo odakle je i uzet. Dakle, sada je kao element drugog niza ono što mi tražimo:

Sada se prelazi na novi iteraciju tj. inkrementriraju se registri Rx i Ry, a dekrementira se registar F4 (on je ustvari counter petlje).

Naravno, na kraju je BNEZ naredba grananja koja odlučuje da li će se petlja još vrteti ili se obavila dovoljan broj puta.

Zaključujemo:

Odsustvo instrukcije za kontrolu petlje u skalarnom kodu, postoji zavisnost po podacima.

Sada možemo napisati i vektorski kod koji će obaviti isti posao.

LD	F0,A
LV	V1,Rx
MULSV	V2,F0,V1
LV	V3,Ry
ADDV	V4,V2,V3
SV	Ry,V4

Kod vektorskog koda je moguće ostvariti 10–20 puta brže izvršenje u odnosu na skalarni kod.

Kritično kod vektorskog procesora je način organizacije memorije i način smeštanja podataka u memoriju. Memorija se sastoji od niza memorijskih banaka (može im se paralelno pristupati) tj. ne mora se čekati novi ciklus za naredni pristup memoriji. Vektorski kod ima samo 6 instrukcija naspram približno 600 instrukcija koje se treba izvršiti u petlji sekvencijalnog koda. Brzini, naravno, doprinosi i odsustvo segmenta koda za kontrolu uslova petlje i dekrementiranje projača. Ove naredbe su u skalarnom kodu skoro 50% od ukupnog broja naredbi.

S obzirom na postojanje zavisnosti po podacima, ADDD naredba čeka da se izvrši MULTD, SD čeka na ADDD, a sve su to instrukcije sa dugim vremenom izvršenja. Dakle, dosta vremena se gubi. Ovaj problem se najčešće rešava odmotavanjem petlje i preuređenjem tako dobijenog koda u skladu sa osnovnim principom, a to je: smanjenje neophodnih ciklusa zastoja u toku izvršenja programa. Naravno, ovaj vid optimizacije ne donosi za sobom smanjenje broja naredbi u skalarnom kodu, već samo optimizuje postojeći broj naredbi.

Naravno, i u vektorskom kodu postoji ta zavisnost (zavisnost između MULTD i ADDD), ali ona će prouzrokovati samo 1 zastoj na svakih 64 elemenata. Na suprot ovom slučaju, u skalarnom kodu će se ovaj zastoj javiti 64 puta za 64 elementa. Dakle, poboljšanje je više nego očigledno. Dakle, kao zaključak možemo naglasiti da vektorski kod može da se izvrši 20–30 puta brže od sekvencijalnog koda.

Način organizacije memorje i način smeštanja podataka u memoriju su kritični. Memorija je organizovana kao niz memorijskih banaka, pa se tim bankama može pristupiti istovremeno (tj. ako imamo samo jedan procesor, on može u svakom ciklusu takta da inicira pristup svakoj sledećoj banci (a ne da čeka da se pristup završi tj. da prođe negde oko 20 ciklusa takta)

NPR: Imamo matricu dimenzije 4x4 i 4 memorijske banke. Elemente matrice možemo smestiti na različite načine zar ne? Naprimer ovako !

M0	M1	M2	M3
A 00	A 01	A 02	A 03
A 10	A 11	A 12	A 13
A 20	A 21	A 22	A 23
A 30	A 31	A 32	A 33

Šta ako se u sukcesivnim memorijskim trenucima zahteva pristup celoj vrsti matrice. Neće doći do konflikata zato što se elementi vrste nalaze u različitim memorijskim bankama. Takođe, ako bi aplikacija zahtevala pristup dijagonalnim elementima ne bi došlo do konflikta. Međutim, ako se zahteva istoveremeni pristup svim elementima jedne **kolone** tada dolazi do konflikata, pa bi latentnost memorije bila vidljiva za svaki element (za pristup jednoj vrsti, konflikt bi bio vidljiv za svaki element vrste jer se oni nalaze ujednoj memorijskoj banci). Ovaj konflikt je moguće razrešiti različitim smeštanjem elemenata. Kao na primer na sledeći način:

M0	M1	M2	M3
A 00	A 01	A 02	A 03
A 13	A 10	A 11	A 12
A 22	A 23	A 20	A 21
A 31	A 32	A 33	A 30

Pristup koloni ili vrsti sada neće dovesti do konflikata, ali će pristup elementima dijagonale dovesti do konflikta. Da bi smo mogli da i ovaj konflikt da rešimo potrebne su nam 5 memorijske banke.

M0	M1	M2	M3	M4
A 00	A 01	A 02	A 03	
A 13		A 10	A 11	A 12
A 21	A 22	A 23		A 20
	A 30	A 31	A 32	A 33

Prvi uslov da bi pristup podacima mogao da bude bez zastoja je da broj memorijskih banaka mora da bude veći od latentnosti memorijskog sistema. Realno, broj banaka je neki stepen dvojke. Postavlja se pitanje koliki je minimalan broj banaka da bi se obezbedio pristup bez konflikata.

Procesor u sukcesivnim vremenskim trenucima inicira pristup bankama, pa onda, broj memorijskih banaka mora biti veći od latentnosti memorijskog sistema, da ne bi došlo do zastoja kada pristupamo sukcesivnim elementima vektora (latentnost memorije je vremenski interval između 2 pristupa memoriji).

M0	M1	M2	M3
A 1	A 2	A 3	A 4
A 5	A 6	A 7	A 8
A 9	A 10	A 11	A 12
A 13	A 14	A 15	A 16

NPR: Za latentnost memorijskog sistma od 5 ciklusa takta, i ako pritom imamo 4 memorijska modula. Pitamo se mi kada će moći da se pristupi elementu a5?

Dakle, treba da protekne 5 memorijskih intervala da bi smo mogli da pristupimo a5. Znači imaćemo zastoj od 2 ciklusa takta za pročitavanje petog elementa. Ovi ciklusi kašnjenja bi rasli ukoliko bi nastavili da čitamo do a16.

Ali ako se koristi veći broj banaka od latentnosti memorije onda nećemo imati ove probleme jer će u isto vreme moći da se pristupi svim podacima (rezličite banke).

Ako se elementi pak nalaze na susednim memorijskim lokacijama broj banaka treba da bude veći od latencije sistema da ne bi došlo do konflikta.

Ali, ako se pristupa a0,a6,a12,a18 doći će do konflikta. Zbog toga uslov uslov da broj banaka bude veći od latencije sistema ne garantuje da neće doći do konflikta. Ako se pristupa elementima sa nekim korakom (STEP K) taj korak se zove **vektorski korak** ili **pomeraj** (vector stride) onda konflikti mogu da nastanu bez obzira što je uslov koji smo malopre dali ispunjen.

Postavlja se pitanje koji je broj banaka potreban da bi se izbegli konflikti ako se elementi učitavaju sa korakom K>1.

Sledeća formula jasno definiše kada će doći do konflikta:

$$\frac{NZS(K, broj\_banaka)}{k} < latentnost\_memorije$$

NPR: Pretpostavimo da imamo 16 memorijskih banaka sa latentnošću od 12 ciklusa takta. Koliko vremena će biti potrebno da se napuni vektorski registar sa 64 elementa ako se elementi koji se pribavljaju nalaze na:

a) rastojanju od K=1

b) rastojanju od K=32

Rešenje:

a)

K=1

Dakle, 12+64=76 ciklusa takta

b)

K=32

$$\frac{NZS(32,16)}{32} < 12?$$

$$\frac{32}{32} = 1$$

Dakle, konflikti postoje! A, koliko je vremena potrebno za pribavljanje 64 elementa.

Pošto se svi elementi nalaze u istom memorijskom modulu za svaki ćemo imati konflikt. Dakle, ukupno vreme će biti: 12*64=768

Da do konflikta ne dođe, potrebno je da K (korak) i broj banaka budu međusobno prosti brojevi. Na primer: (16 i 5).

Vektorizacija petlje

Da bi vektorska mašina mogla da se koristi neophodno je posedovanje **vektorizirajućih kompilatora** ili viših programskih jezika. U drugom slučaju deo koda koji može da se izrazi u vektorskom kodu određuje programer a u prvom slučaju kompajler. Dakle, u prvom slučaju programer piše klasičan program a kompajler takav kod prevodi na vektorski kod.

Da bi kompajler mogao da utvrdi koji deo programa može da se izvrši u vektorskom obliku on mora da utvrdi koje zavisnosti postoje između podataka. Jedini kandidati za vektorizaciju su petlje, i to one u kojima se obrađuje elementi nekog polja. Osnovni zadatak kompajlera je da utvrdi da li se petlja ili deo petlje može vektorizovati. U slučaju da postoji više **ugnježdenih petlji** vrši se vektorizacija najdublje ugnježdene petelje. Kod paralelizacije se koristi spoljnja petlja.

Kompajler treba da utvdi koje zavisnosti postoje između promenljivih u telu petlje. Zadržaćemo se na **RAW** zavisnostime jer **WAR** i **WAW** ne otežavaju vektorizaciju jer se mogu izbeći preimenovanjem registara. Dakle, zavisnosti kojima se mi bavimo su tipa **RAW**.

Sada razmatramo petlju:

```
Do 10 i=1,100
S1:   A(i+1) = A(i) + B(i)
S2:   B(i+1) = B(i) * A(i+1)
Do CONTINUE
```

Moguće su sledeće varijante zavisnosti koje se javljaju:

- Da se u S1 koristi rezultat te iste instrukcije iz neke predhodne iteracije
- Da se u S1 u tekućoj iteraciju koristi rezultat S2 iz predhodne iteracije
- Da se u instrukciji S2 koristi rezultat tekuće iteracije instrukcije S1

Do ovih zavisnosti dolazi zato što se radi o **dubokoj protočnosti**. Sledeća iteracija krene sa izvršenjem a da predhodna još uvek nije završena. Zbog toga dolazi do RAW hazarda koga vektorski računar ne proverava ako je stavljena vektorska instrukcija. Dakle, ako imamo neku od prve dve nabrojane situacije doćiće do RAW zavisnosti koju vektorski računar ne proverava.

Zbog toga petlje koje sadrže ovakve zavisnosti ne mogu biti vektorizovane. Dakle, naš programski kod ne bi mogao da se vektorizuje.

LOOP CARRIED zavisnosti nastaju kao posledica korišćenja rezultata predhodne iteracije, sprečavaju vektorizaciju tako da je osnovni zadatak kompilatora da utvrdi da li postoje **LOOP CARRIED** zavisnosti.

Dakle zavisnosti pod 1 i 2 su takozvane **loop carried** zavisnosti i one su te koje prave problem.

I=1	A(2)=(A(1) + B(1))
	B(2)=(B(1) * A(2)
I=2	A(3)=(A(2) + B(2))

	B(3)=(B(2) * A(3)

Osnovni zadatak kompajlera je da proveri da li postoje Loop Carry zavisnosti i u slučaju detekcije takve zavisnosti, kompajler zaključuje da petlju ne može vektorizovati.

U slučaju da programer piše vektorsku naredbu, on tada garantuje da je ona ispravna tj. nema zavisnosti i tada je kompajler ne proverava. Dakle, ako pišemo vektorski kod, tada se on ne proverava, a kada kompajler vektorizuje sekvencijalni kod onda se vrši provera.

Testiranje

Osnovni zadatak kompajlera je da utvrdi zavisnosti podataka (Loop Carry zavisnosti). **NZD** ili **GCD test** utvrđuje ove zavisnosti.

NZD (GCD) test je vezan za iterativne indekse. Pretpostavimo da se vrši upis u element polja kada je on indeksiran sa **a*I + b** pri čemu je **I** ustvari taj iterativni indeks, a **a** i **b** su konstante. Takođe pretpostavimo da se tom elementu pristupa radi čitanja kada je on indeksiran sa **c*I + d** gde su **c** i **d** takođe konstante.

Loop Carry zavisnost postoji ako postoje dva iterativna indeksa **J** i **K** (oba unutar granica petlje, a oba indeksitaju elemente u različitim iteracija) tako da se u petlji pamti rezultat u element kada je indeksiran sa **c*K + d** (pri čemu je J <> K), ako važi:

a*J + b = c*K + d

Dakle, ukoliko je ispunjena ova formula **Loop Carry** zavisnost postoji.

NZD test se zasniva na činjenici da ako Loop Carry zavisnost postoji tada NZD(c,a) mora da deli bez ostatka d–b za svako (J <> K). Ovaj test je dovoljan da garantuje da u petlji ne postoje loop carry zavisnosti. Međutim može da se desi da test prođe i da registruje zavisnosti a da stvarne zavisnosti ustvari nema. To je zato što ovaj test ne uzima u obzir **granice petlje**.

Na primer:

Korišćenjem NZD testa utvrditi da li postoji zavisnost u sledećoj petlji?

```
Do 10 i=1,100
X(2*i + 3) = X(2*i)*0.5
10 Continue
```

a=2

b=3

c=2

d=0

Dakle, NZD(2,2)=2

d–b=0–3=–3

Zaključak: 2 ne deli –3 bez ostatka dakle, petlja može da se vektorizuje.

Sada bi mogli i praktično da proverimo da li je to tako! Da bi bili 100% sigurni:

Za i=1

X(5)=x(2)*0.5

Za i=2

X(5)=x(4)*0.5

Za i=3

X(9)=x(6)*0.5

Dakle, zavisnosti stvarno nema !

Da bi NZD test mogao da se primeni petlja mora da bude **normalizovana**, to znači da indeks u petlji mora da počinje od 1 a da se on inkrementira za +1 u svakoj iteraciji.

Na Primer:

Do 10 i,2,100,2

X(50*i+1)=x(i–1)*k

10 Continue

Ova petlja iako ima 100 indeksa ima samo 50 iteracija. Dakle, ovo nije normalizovana petlja. Normalizovana petlja bi bila:

Do 10 i,1,50

X(100*i+1)=x(2i–1)*k

10 Continue

Sada je početna petlja normalizovana i sada možemo da primenimo NZD test. Dakle, a=100, b=1, c=2, f=–1. Dakle, sada bi trebali da primenimo NZD test.

NZD(a,c)=NZD(100,2)=2

d–b=–2

Dakle, 2 deli –2 bez ostatka. To znači da je detektovana loop carry zavisnost prema ovom testu. Međutim ukoliko proverimo da li te zavisnosti zaista ima, utvrdili bi smo da je nema. Hajde sada to i da praktično pokažemo:

Za i=1 x(101)=x(1)*k

Za i=2 x)201)=x)3_*k

Za i=50 x(5001)=x(99)*k

Za i=51 x(5101)=x(101)*k

Primećujemo da loop carry zavisnosti stvarno nema jer test ne uzima u obzir granice petlje. Dakle, ukoliko bi petlju izvrteli još samo jedared nastala bi zavisnost što je i pokazano crvenom bojom.

Dakle, zaključićemo da ukoliko bilo koji test da rezultat da zavisnosti postoje, ne treba zdravo za gotovo odustati od vektorizacije, ali ako on da rezultat da zavisnosti nema, onda nema šanse da je pogrešio, već možemo biti sigurni da je vektorizacija moguća. Naravno, postoji i mogućnost da se promenom redosleda naredbi izbegnu zavisnosti. Naravno, za tu svrhu može poslužiti i razbijanje petlje.

Primer:

U ovoj petlji postoji Loop Carry zavisnost.

```
DO 10 i=1,N
S1: x(i)=y(i–1)*z(i)
S2: y(i)=2*y(i)
10 Continue
```

Zaista, u ovoj petlji postoji zavisnost popromenljivoj y, međutim ova zavisnost može biti otklonjena ako preuredimo instrukcije ili pak razbijemo početnu petlju na dve. Jednu u kojoj se računa y, i drugu u kojoj se računa X.

To bi moglo ovako da izgleda:

```
DO 10 i=1,N
Y(i)=2*y(i)
10 Continue

DO 20 i=1,N
X(i)=y(i–1)*Z(i)
20 Continue
```

Sada su jasno razdvojene petlje koje se mogu vektorizovati. Vektorske varijante ovih petlju bi bile:

```
Y(1:N) = 2*y(1:N)
X(1:N) = y(0:N–1)*Z(1:N)
```

Nravno, nije uvek moguće upotrebiti ovaj **trik**. Naime, da smo imali situaciju u kojoj S1 zavisi od S2 i da S2 zavisi od S1 onda nikako ne bi mogli ni da promenimo redosled instrukcija niti da razbijemo petlju na dve. Takva petlja se ne može vektorizovati.

Pored Loop Carry zavisnosti mogu se javiti i prave (zavisnosti RAW zavisnosti), antizavisnosti (WAR) i izlazne zavisnosti WAW. Antizavisnosti i izlazne zavisnosti ne sprečavaju vektorizaciju!

Kao ilustraciju ovih zavisnosti možete pogledati sledeći primer, u kome ćete morati da pronađete sve prave zavisnosti (RAW), antizavisnosti i izlazne zavisnosti. Zatim, eliminisati antizavisnosti i izlazne zavsinosti preimenovanjem!

```
DO 10 i=1,100
1:    y(i)=x(i)/s
2:    x(i)=x(i)+s
3:    z(i)=y(i)+s
4;    y(i)=s–y(i)
```

10: Continue

RAW zavisnosti su između 1–3 naredbe i između 1–4 naredbe

WAR zavisnosti su između 1–2 naredbe i između 3–4

WAW zavisnosti su između 1–4

Neke od zavisnosti se mogu eliminisati preimenovanjem tj. neće imati uticaj na vektorizaciju programa. Tj. ako izvršimo preimenovanja:

Y(i) –> T(i)

X(i) –> x1(i)

Onda početna petlja postaje:

DO 10 i=1,100

1: T(i)=x(i)/s

2: x1(i)=x(i)+s

3: z(i)=T(i)+s

4; y(i)=s–T(i)

10: Continue

Posle ove transformacije (preimenovanja) i dalje postoje prave zavisnosti. Ali smo uspešno eliminisali sve ostale. Naravno, **antizavisnosti** i **izlazne zavisnosti** su otklonjene preimenovanjem.

Vektor maske:

Kod vektorizacije petlji gde se javljaju naredbe uslovnog grananja koristi se **vektor maske**. NPR:

DO 10 i=1,64

IF (a(i).ne.d) THEN

B(i)=b(i)/a(i)

End if

10 continue

Vektorski kod za ovu petlju bi izgledao ovako:

LV V1,a

LV V2,b

LD F0,#0

LV naredba je naredba za učitavanje vektora (Load vector).

VM je vektor maske i ima onoliko elementa koliko ima i elemenata u vektorskom registru). U vektoru maske mogu biti nule i jedinice. **Nule** znače da element nije dostupan, a **jedinice** da se nad njim može izvršiti odgovarajuća instrukcija.

LV V1,A

LV V2,B

LD F_0,#0

SNESV V1,F_0

DIV V

SU

CVM

Sve opracije koje slede SNESU, izvršavaju se pod maskom. Pa u ovom konkretnom slučaju imamo deljenje pod maskom (DIVV V2,V2,V1).

CVM naredba služi za brisanje maske, tj. svi bitovi se postave na 0

Analiza zavisnosti sa više ugnježdenih petlji

Tada pristup elementu polja zavisi od više indeksa.

DO 10 $I^1=e^1,u^1$

DO 10 $I^2=e^2,u^2$

DO 10 $I^3=e^3,u^3$

.

.

DO $10^n=e^n,u^n$

$S_1(I)$

$S_2(I)$

.

.

$S_n(I)$

10 Continue

I^1,.....I^n su iterativni indeksi petlje

E^i, u^i su granice petlje

I je uređena n–torka indeksa

S su naredbe dodeljivanja

Indeksi I moraju da zadovoljavaju leksikografsku uređenost tj. mora da važi za

I=(4,3,1) i J=(3,4,5) važi da je:

I > J

Postupak **vektorizacije ugnježdenih petlji** može se zapisati u vidu nekoliko koraka.

Prvi korak je da se uoče svi parovi generisanih i korišćenih promenljivih u petlji, a onda za svaki takav par odrediti vektor zavisnosti d na sledeći način:

Ako je generisana promenljiva X indeksirana sa I, pri čemu je f neka celobrojna funkcija indeksa nad indeksima I1,I2....In i korišćena promenljiva X(g(I)) tada se vektor zavisnosti d izračunava kao:

D=j(i)–g(i)

NPR:

X(f(I))

X(g(I))

=>

d=j(I)–g(I)

Treći korak je formiranje **matrice zavisnosti** po podacima.

D1,d2,d3....dk => D = (d1 d2 dk)

Ovde je D matrica zavisnosti. Na osnovu matrice zavisnosti možemo da utvrdimo po kojoj indeksnoj promenljivoj je moguće izvršiti vektorizaciju a po kojoj ne.

Primer: Pronaći sve vektore zavisnosti u sledećem gnezdu petlje !

```
DO 10 i=1,5
DO 10 j=1,10
DO 10 k=1,20
     A(i,j,k)=A(i–1,j,k+1) + B(i,j,k)
     B(i,j,k+1) = B(i,j–1,k–1)*3
10 Continue
```

A: $d1=(i,j,k+1)^T - (i-1,j,k+1)^T = (1,0,-1)^T$ -> (<,=,>)

B: $d2=(i,j,k+1)^T - (i,j,k)^T = (0,0,1)^T$ -> (=,=,<)

$d3=(i,j,k+1)^T - (i,j-1,k-1)^T = (0,1,2)$ -> (=,<,<)

Matrica sada izgleda ovako:

$$D=\begin{bmatrix} < & = & = \\ = & = & < \\ > & < & < \end{bmatrix}$$

$$D=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ -1 & 1 & 2 \end{bmatrix}$$

Nije bitno da redosled vrsti bude baš ovakav. Možemo pisati i prvo K, pa J pa tek onda I. Naravno, obično se piše I,J,K.

Paralelizacija je moguća po onoj indeksnoj promeni koja odgovara vrsti matrice zavisnosti D sa svim nulama.

Vrednost – tj. negativna vrednost ukazuje na antizavisnost

Vrednost **0** ukazuje na istu iteraciju

Vrednost **+** ukazuje na loop carry zavisnost.

Što se tiče pravca zavisnosti:

$$pravac\_zavisnosti=\begin{cases} <, & d_i>0 & ukazuje\_na\_loop\_carry\_zavisnost \\ =, & d_i=0 & prava\_RAW\_zavisnost \\ >, & d_i<0 & Antizavisnost \end{cases}$$

Vektorizacija nije moguća po onoj indeksnoj promenljivoj (vrsti) po kojoj jednovremeno postoje pravci unapred i unazad dakle, < i > tj. tada postoji **loop carry** zavisnost.

Primer:

```
I=1
J=1
```

K=1

A(1,1,1)=A(0,1,2)+B(1,1,1)

B(1,1,2)=B(1,0,0)

ZA K=2 A(1,1,2) + A(0,1,3) + B(1,1,2)

B(1,1,3) = B(1,0,1)

K=3

.....

Elementarne transformacije:

Elementarne transformacije se primenjuju nad I,J,K da bi izvršili vektorizaciju petlje. Postoje 3 elementarne transformacije koje se mogu obaviti nad indeksnim skupovima.

- Permutacija
- Inverzija (obrtanje)
- Krivljenje (jednog iterativnog indeksa u odnosu na drugi)

Sve te elementarne transformacije se opisuju preko **matrica transformacije**:

I1=l1,n1

I2=l2,n2

.

.

.

Im=lm,nm

Matrica transformacije je kvadratna matrica dimenaija mxm, za koju mora da važi sledeće da bi bila validna:

T_{MxM} Vrši preslikavanje jednog indeksnog skupa u drugi (mora da bude kvadratna)

Abs (det(T)) = 1 Apsolutna vrednsot determinante mora da bude 1.

Polazi se od jedinične matrice mxm i u njoj se vrše modifikacije.

Permutacija:

Ij Ik

U jediničnoj matrici mxm se zamene mesta j–toj i k–toj vrsti tako da se dobija matrica T.

I=1,n

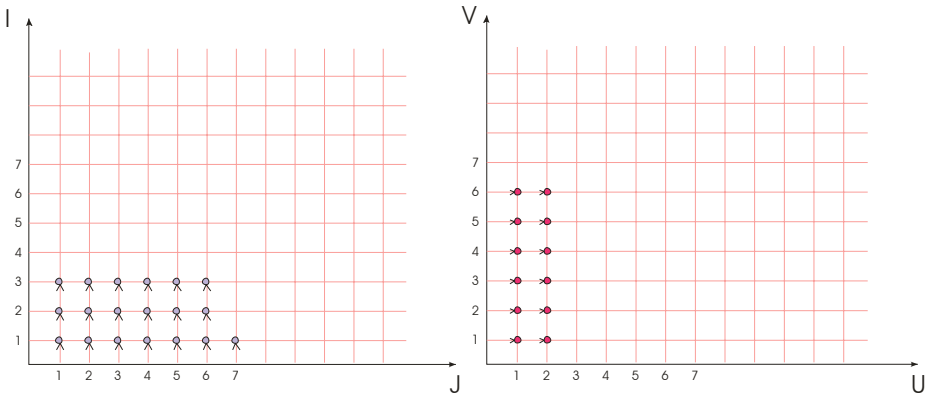
J=1,m

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} U \\ V \end{bmatrix} = T \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} J \\ I \end{bmatrix}$$

U=j u=1,m

V=I v=1,n



Transformacija ne sme da menja smer zavisnosti!!!!!

$$\overline{d} = T \cdot d$$

$$d = (d_1, d_2, ..., d_i, ...d_m)$$

Sa ovako definisanim vektorom d, vrši se poređenje sa (0,0,0,0,0,.....0)

d>0: Znači da je prvi nenulti element u vektoru d veći od nule.

Na primer:

d_1=(0,1,−1,2) dakle, sada je $I_1 > I_2$

d_2=(0,0,−3,5) dakle, sada je $I_1 < I_2$

d_3=(1,−1,0) dakle, sada je $I_1 > I_2$

d_4=(−5,7,3) dakle, sada je $I_1 > I_2$

Primer primenjivanja permutacije:

```
Do i=1,n
Do j=1,m
    A(j)=A(j)+C(i,j)
```

Ova petlja postaje:

```
Do U=1,m
Do V=1,n
    A(U)=A(U)+C(V,U)
```

$$\begin{bmatrix} U \\ V \end{bmatrix} = T \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} J \\ I \end{bmatrix}$$

U=j

V=i

Obrtanje

U jediničnoj matrici u vrsti koja odgovara tom iterativnom indeksu vrednost dijagonalnog elementa je –1.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\begin{bmatrix} U \\ V \end{bmatrix} = T \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} I \\ -J \end{bmatrix}$$

U=i

V=–j

Primer:

Do i=1,n

Do j=1,n

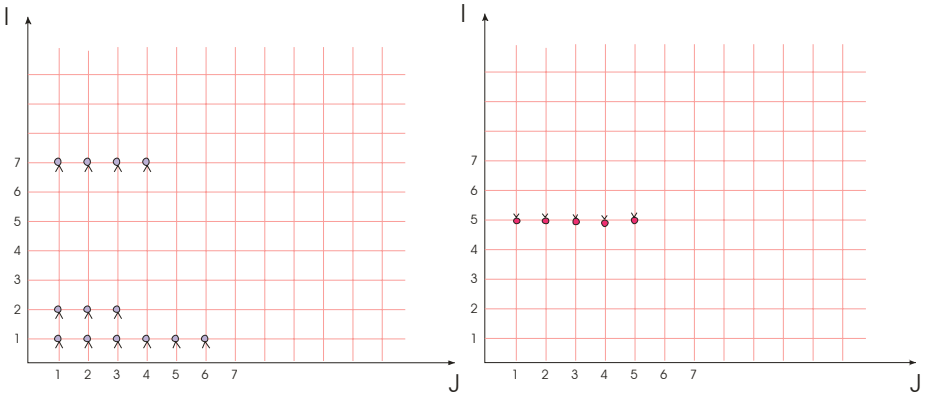
A(i,j)=A(i–1,j+1)

Ova petlja posle transformacije postaje:

Do U=1,n

Do V=–n,–1

A(U,–V)=A(U–1,–V+1)



Krivljenje

Krivljenje jednog iteracionog indeksa u odnosu na drugi.

$(p_1, p_2, \ldots\ldots p_i, \ldots\ldots p_{j-1}, p_j, p_{j+1}, \ldots\ldots p_k)$

Ako hoćemo da primenimo krivljenje indeksne promenljive J u odnosu na neki faktor f, tada treba vršimo sledeću transformaciju:

$(p_1, p_2, \ldots\ldots p_i, \ldots\ldots p_{j-1}, f*p_i+p_j, p_{j+1}, \ldots\ldots p_k)$

Iterativni indeks koji krivimo postaje zavistan od iterativnog indeksa u odnosu na koji ga krivimo. Na primer:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} i \\ j \\ k \end{matrix}$$ ako krivimo K u odnosu na i $$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ f & 0 & 1 \end{bmatrix}$$

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} i \\ j \\ k \end{matrix}$$ ako krivimo K u odnosu na j $$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & f & 1 \end{bmatrix}$$

Na primer:

Do i=1,n

Do j=1,n

A(i,j)=A(i,j−1)+A(i−1,j)

Ako ovu petlju transformišemo tako što ćemo kriviti J u odnosu na I za faktor 2, onda bi dobili:

$T = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix}$ Važno je napomenuti da se uvek vrši krivljenje nekog unutrašnjeg u odnosu na neki spoljašnji iterativni indeks.

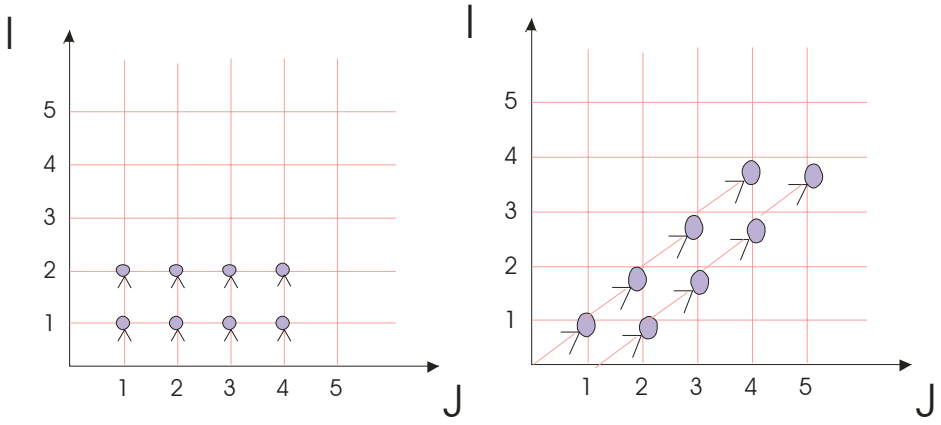
$$\begin{bmatrix} U \\ V \end{bmatrix} = T \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} I \\ 2I + J \end{bmatrix}$$

U=i U=1,n

V=2i+j V=2n+1,2n+n

J=V−2U

A(U,V−2U)=A(U,V−2n−1)+A(U−1,V−2U)



Ove elementarne transformacije se zovu unimodularne i imaju osobinu da kompozicija elementarnih preslikavanja takođe daje validne matrice transformacije. Dakle, možemo primenjivati ove transformacije koliko god puta hoćemo i u kom kod redosledu hoćemo, matrica koju dobijemo biće validna.

SIMD računari

Alternativan način za ubrzanje vektorskih izračunavanja je pomoću **SIMD** računara. Za razliku od vektorskih računara koji koriste vremenski paralelizam, kod SIMD računara se koristi **prostorni paralelizam** tako što postoji više identičnih procesnih elemenata (PE) koji su jednovremeno aktivni i koji izvršavaju istu instrukciju ali nad različitim podacima.

SIMD računar je sinhrono polje procesnih elemenata čijim radom upravlja CU (Control Unit). U datom trenutku svi procesni elementi u polju izvršacvaju istu instrukciju, ali nad različitim podacima. To znači da se uz pomoć SIMD računara postiže paralelizam u obradi podataka, a ne u izvršenju instrukcija. I SIMD računari su kao i vektorski specijalno projektovani da ubrzaju vektorska izračunavanja i dodaju se običnom procesoru za ubrzavanje vektorskih izračunavanja.

SIMD računari se javljaju u dve arhitekturne grupa:

- Procesorska polja koja koriste RAM memorije
- Asocijativni procesori koji koriste asocijativne memorije (sadržajno adresirane)

Procesorska polja koja koriste RAM memorije su se pokazala bolja za veći broj aplikacija, pa ih koristi najveći broj danas raspoloživih SIMD računara.

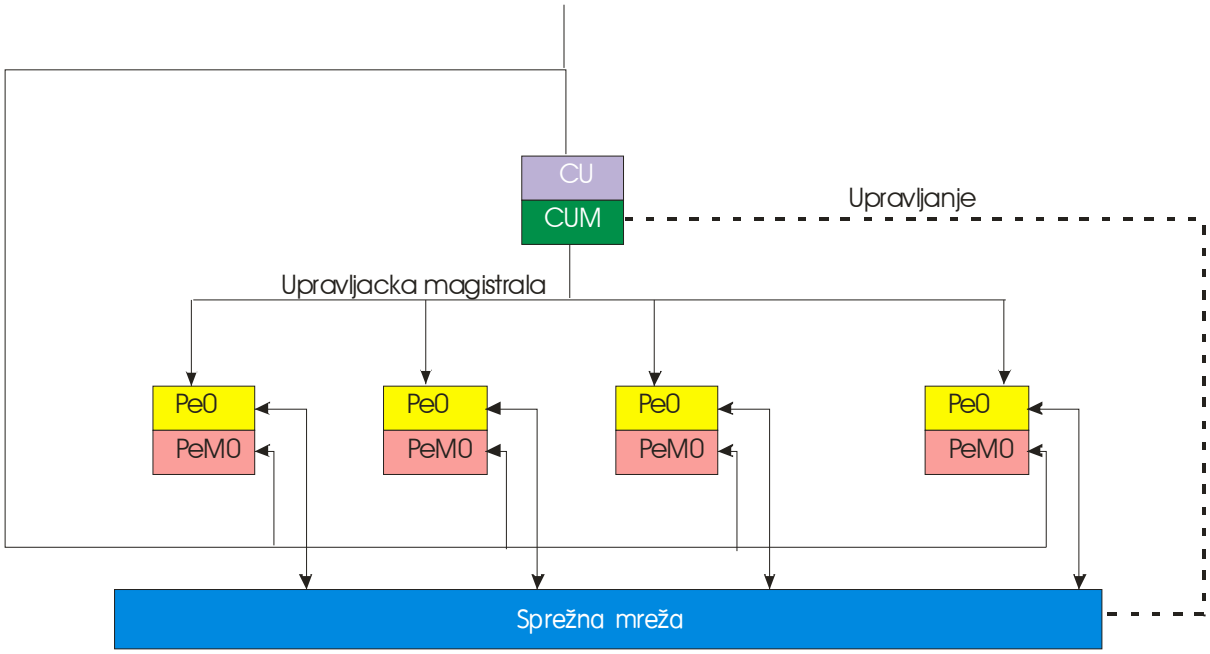
SIMD računari koji koriste RAM memorije

Kada su u pitanju procesorska polja koja koriste RAM memorije moguće su dve konfiguracije:

- Procesorska polja sa distribuiranom memorijom
- Pocesorska polja sa deljivom memorijom

Broj procesnih elemenata u sistemu je uvek stepen dvojke: 2^m

Procesorska polja sa distribuiranom memorijom



Svi procesni elementi imaju svoju **loklnu memoriju**. Radom sprežne mreže upravlja CU. Dakle, ova konfiguracija se sastoji od $n=2^m$ sinhronih procesnih elemenata čijim radom upravlja jedinstvena CU. Svaki procesni element je ALU jedinica sa pridruženim radnim registrima i lokalnom memorijom koja sadrži distribuirane vektorske podatke nad kojima će taj procesni element vršiti neka izračunavanja. Naravno, i CU ima svoju lokalnu memoriju koja služi za smeštaj instrukcija i skalarnih podataka. Sistemski i korisnički programi se izvrešavaju pod direknim upravljanjem CU. Korisnički programi se pune u lokalnu memoriju. Zadatak CU je da odredi gde će se svaka instrukcija izvršiti (i da dekodira instrukciju). Instrukcije **upravljačkog tipa** i **skalarnog tipa** se izvršavaju direno u upravljačkoj jedinici (CU), dok se **vektorske instrukcije** izvršavaju u polju procesnih elemenata. Vektorske instrukcije se smeštaju u procesne elemente preko upravljačke magistrale.

U jednom trenutku CU emituje jednu instrukciju i svi procesni elementi izvršvaju tu instrukciju, ali nad podacima iz svoje lokalne memorije. U ovom procesnom elementu pored aritmetičko logičkih instrukcija se mogu izvršavati i instrukcije **rutiranja** i **maskiranja**.

Rutiranje su instrukcije koje omogućavaju razmenu podataka između procesnih elemenata. Funkcije rutiranja mogu da obavljaju **Broadcast** ili **selektivnu** emisiju.

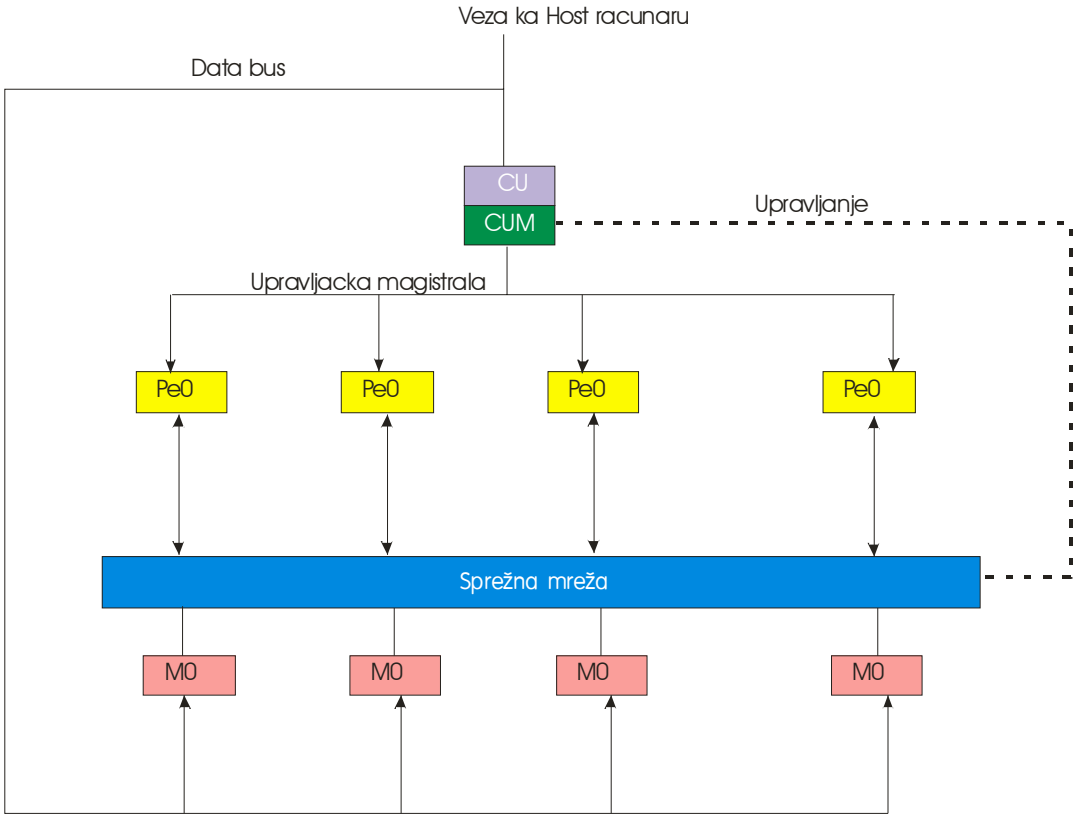
Vektorski operandi se distribuiraju u lokalne memorije procesnih elemenata od strane **HOST** računara preko magistrale podataka i to pre početka izvšenja programa. Moguće je neke podatke poslati procesnim elementima i preko upravljačke magistrale iz CU–a (ako je to neki skalarni podatak).

Za upravljanje statusom svakog procesnog elementa koristi se **maskiranje**. Za vreme izvršenja jedne vektorske instrukcije ne moraju svi procesni elementi u polju biti aktivni. Maskiranjem se skup procesnih elmenata deli na podskup **aktivnih** i podskup **pasivnih** procesnih elemenata.

Zadatak **Host** računara je da obavlja ulazno izlazne aktivnosti za procesorsko polje, da obavlja kompletiranje i LOAD–ovanje korisničkih programa u CU memoriju i da izvršava druge sistemske programe. Razmena podataka između procesorskih elemenata se ostvaruje preko **sprežne mreže**. primer ILLIAC IV (64 procesnih elemenata).

Pocesorska polja sa deljivom memorijom

Ova druga arhitekturna konfiguracija se razlikuje od prve po dvama stvarima:



Distribuirani memorijski moduli su zamenjeni **deljivim** memorijskim modulima kojima mogu pristupati svi procesni elementi u sistemu preko sprežne mreže.

Druga razlika je funkcija sprežne mreže. Kod prve konfiguracije je bilo povezivanje procesnih elemenata međusobno, a sada je povezivanje procesnih elemenata sa određenim memorijskim modulom. Sprežna mreža je bolja ako može da obezbedi da bez konflikata što veći broj procesnih elemenata može da pristupi memorijskim modulima.

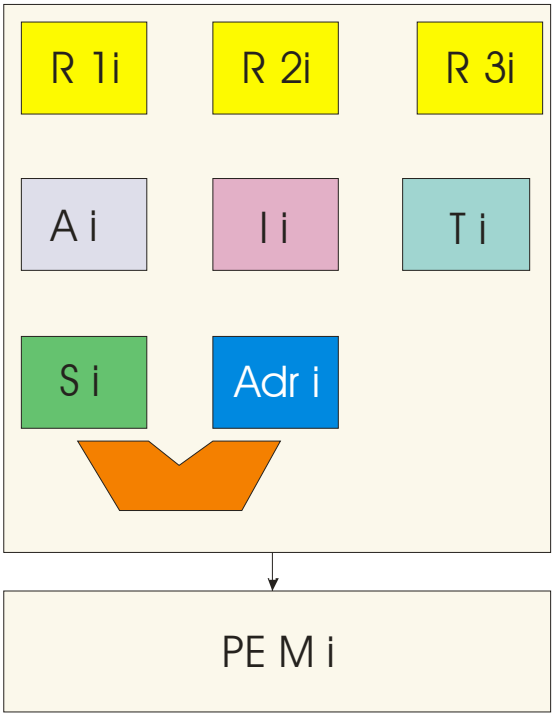
Za kraj poređenja ove dve konfiguracije daćemo primere realnih računara koji ih koriste. Naime, prvu konfiguraciju koristi ILLIAC iV (za 64 procesna elementa), a drugu konfiguraciju koristi BSP (sa N=16 i M=17).

Pokazalo se da je prva konfiguracija bolja od druge. Broj procesnih elemenata može da varira od nekoliko desetina do nekoliko hiljada (računar CM–2 ima 64000 procesnih elemenata, ali su oni jednostavni (jednobitni) procesori). Takođe i složenost procesnih elemenata može da varira (ko ILLIAC–a su procesori 64–bitni).

MasPar MP–1 se sastoji od 1024–16K procesnih elemenata pri čemu je na jednom čipu realizovano 32 procesna elementa, a kod MC–2 168 procesnih elemenata po čipu.

Struktura procesnih elemenata

Uzmimo prvu konfiguraciju. Jedan procesni element sa te konfiguracije izgleda ovako.



Ai Adresni registar

Ii Indeksni registar

Si Flag registar

R Skup radnih registara

Svaki procesni element ima određeni broj **radnih registara** i to najmanje tri. Skup radnih registara se koristi za pamćenje operanada pribavljenih iz lokalne memorije (dva se koriste za operande a jedan za rezultat) upravo zbog toga je najmanje tri registra.

Ai,Ii registri služe za adresiranje lokalne memorije, pri čemu se u registar Ai pamti **bazna adresa** operanda i tu adresu emituje CU svim procesnim elementima, a Ii je **indeksni registar** koji zajedno sa baznim registrom služi za adresiranje lokalne memorije i njegov sadržaj može da postavi programer. Ovako, sa jednom baznom adresom možemo da pristupimo različitim memorijskim modulima i različitim adresama.

Ti je **transportni registar** i kada se obavlja razmena podataka između procesnih elemenata u suštini se obavlja razmena podataka transportnih registara. Da ne bi došlo do konflikata, postoji zapravo dva Ti–a; jedan za **prijem** a drugi za **slanje** podataka. Veza procesnih elementa sa sprežnom mrežom je u stvari veza za Ti–a sa drugim procesnim elementima. Si je statusni fleg registar i njegov sadržaj određuje da li je procesni element u toku izvršenja date instrukcije tj. aktivan ili ne? Ukoliko je sadržaj statusnog fleg registra 1, on je aktivan, a ukoliko je 0 on je neaktivan.

Svaki procesni element u procesorskom polju je identifikovan jedinstvenom adresom koja se pamti u Adr registru.

Način maskiranja procesnih elemenata u SIMD procesoru

Može se koristiti nekoliko načina **maskiranja**. Ono što je zajedničko za sve maske je da one dele skup procesnih elementata na dva disjunktna podskupa. (**aktivni** i **pasivni** elemnti). Kod nekih načina maskiranja je moguće jednim maskom izdvojiti proizvoljan podskup procesnih elementata, dok negde moramo poštovati neka pravila (ne može proizvoljno). Sadržaj registra maske se može definisati u fazi prevođenja programa (to je **statičko**), a ako se definiše u toku izvršenja programa, onda je to **dinamičko** maskiranje.

Direktno maskiranje

Tu postoji jedan vektor maske koji ima onoliko bitova koliko ima procesnih elemenata u sistemu: (VM –vektor maske).

VM(i)=PEi (i–ti bit odgovara i–tom procesnom elementu). Ovo je bilo zgodno kod ILLIACA jer nije imao mnogo procesnih elemenata, pa nije bio mnogo dugačak VM. Sadržaj VM–a u fazi prevođenja

programa definiše kompilator. Kada se izvršava instrukcija CU (control unit) zajedno sa instrukcijom emituje u VM masku, naravno, preko upravljačke magistrale.

Dobra strana ovog načina maskiranja: Jednom maskom se može maskirati proizvoljan broj procesnih elemenata; ne zahteva se nikakva regularna struktura. Loša osobina je to što ukoliko imamo više procesnih elemenata, VM mora biti vrlo dugačak (sastojaće se od velikog broja Flip Flopova, a i neće moći da se prenese preko standardne upravljačke magistrale.

Adresno maskiranje

U ovom slučaju takođe imamo $N=2^m$ bitova. Maska je m–bitna. I–ti bit ove maske može uzeti vrednost: 0,1 il x (x znači da nije definisano tj. može biti i 0 i 1). I–ti bajt VM–a odgovara i–tom bitu adrese procesnog elementa. Aktivni su oni procesni elementi čija se adresa uparuje (poklapa) sa maskom.

Na primer:

$N=2^3=8$ i neka je maska 1x0, u ovom slučaju biće aktivni oni procesni elementi čija adresa 1 x 0. pošto x može biti ili 0 ili 1, onda će aktivni elementi biti elementi sa adresom 110 i 100. Ovakav načina maskiranja, **ne može** da pokrije odgovarajući broj procesnih elemenata niti proizvoljne procesne elemente. Tako da se često koristi nekoliko maski da bi se pokrili željeni elementi. Ipak, kod mnogih problema postoji prirodna zavisnost, kada koji elementi rade, tako da nije baš toliko nezgodno koristiti ove maske. Ove maske određuje sam programer u toku pisanja programa, a CU (control unit) ih dekodira i emituje procesnim elementima. I ovde, kao i u predhodnom primeru se sadržaj maske definiše statički.

Maskiranje po podacima

Ono predstavlja implicitan rezultat naredbe uslovnog grananja u zavisnosti od podatka koji je lokalni za svaki procesni element. Kad god se emituje naredba uslovnog grananja od strane (control unit–a) ona se izvršava nad podskupom koji je lokalni (nalazi se u lokalnoj memoriji) za dati procesni element. Naredba uslovnog grananja kojom se postiže ovo maskiranje je sledećeg oblika.

WHERE (uslov) do (niz A)

Elsewhere (niz B)

Svaki procesni element testira uslov, nad podatkom iz svoje lokalne memorije, ako j true, onda je 1, a ako je false onda je 0.

Control unit emituje NIZ instrukcija koji smo označili sa A i njih izvršavaju oni procsni elementi kod kojih je rezultat testa WHERE bio 0, naravno, niz instrukcija B će se izvršavati na onim procesnim elementima koji su bili neaktivni za niz naredbi A.

Ovaj način maskiranja se može kombinovati sa predhodno pomenuta dva načina tj. uslov može biti ili adresaili pak nešto drugo.

Priemer:

Neka je potrebno naći sumu S_k prvih K komponenti vektora V_{nx1} koji ima n elemenata.

Sve ove sume za različite vrednosti K nazivaju se prefiks sume.

$$S(k)=\sum_{i=0}^{k} V_i \qquad k=0,1,2,\ldots\ldots n-1$$

Ovaj problem možemo da rešimo ako imamo n procesora, prefiks suma može da se reši u $\log_2 n$ koraka, pri čemu jedan korak obavlja sabiranje i razmenu podataka.

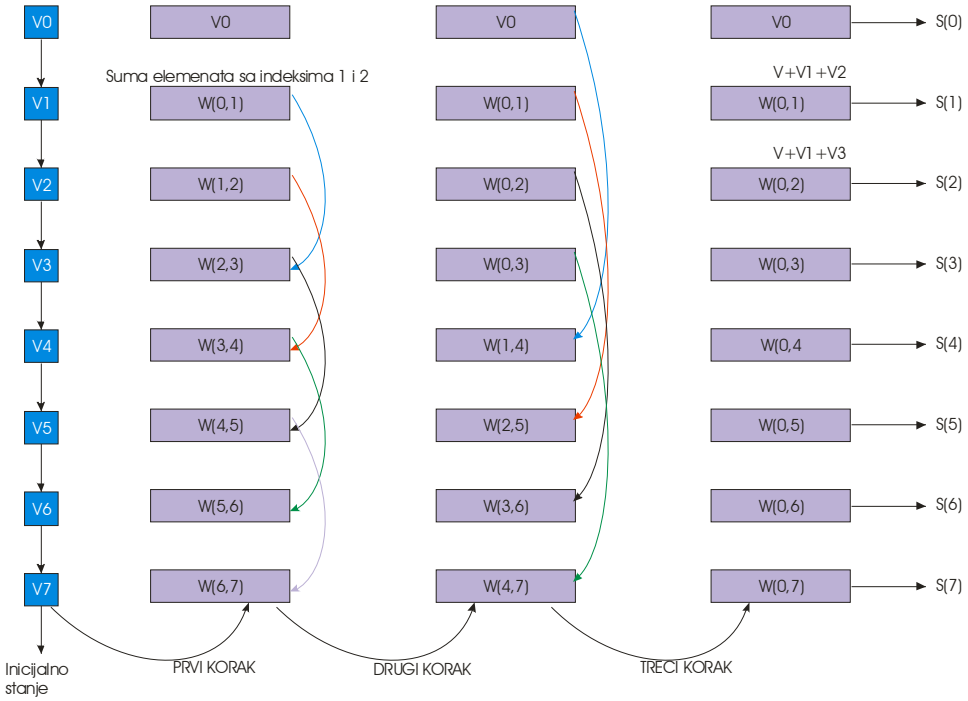
Uzmimo na primer da je n=8

Usvojimo pretpostavku da se u lokalnim memorijama svakog procesnog elementa nalazi i–ti element vektora čiji prefiks sume tražimo. Procesni element sa adresom i šalje podatke u procesni element sa adresom i+1. u ovom koraku PE_7 je maskiran.

U ovom trenutku je, kao što smo zaključili maskiran procesni element PE_7 i tada može da prima podatke, ali ne može da ih šalje. Naravno, kada šalje, procesni element zadržava kopiju toga što je poslao za sebe. Dakle, svaki Procesni element obavlja sabiranje onoga što je bilo u njegovoj lokalnoj memoriji i onog što je dobio. PE0 ne obavlja operaciju sabiranja tj. on je maskiran.

Drugi korak: računa PE_i –> PE_{i+2}. PE_i šalje rezultat sabiranja PE_i+3 dok su $PE_{6,7}$ maskirani.

Treći korak: računa PE_i –> PE_{i+4} dakle, procesni elementi PE_4, PE_5, PE_6 i PE_7 su maskirani.



Sprežne mreže

Procesorska polja se međusobno razlikuju po **sprežnim mrežama**. Iste sprežne mreže se koriste i kod MIMD i kod SIMD računara, samo je raličito upravljanje. Sprežne mreže definišu topologiju jednog procesorskog polja. U odnosu na način rada sprežne mreže se mogu podeliti na **statičke** i **dinamiče**.

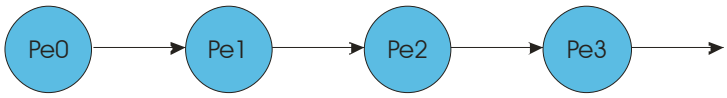
Kod ***statičkih*** sprežnih mreža, veze između procesnih elemenata su fiksne, unpred dodeljene i definisane i ne mogu se menjati, definišu se u fazi projektovanja polja; veze su tipa point–to–point.

Kod ***dinamičkih*** pak, veze između procesnih elemenata se mogu uspostavljati po zahtevu, imaju unapred dodeljene već postojeće komutatore, kojima se po zahtevu mogu ostvariti veze između traženih procesnih elemenata. Oba tipa mreža imamo i kod SIMD i kod MIMD računara.

Statičke sprežne mreže

Kod **Statičke** sprežne mreže problem upravljanja mrežom nije složen. Ovde se javlja podela u odnosu na broj potrebnih dimenzija za povezivanje:

Jednodimenzionalne:



Dijametar sprežne mreže se definiše kao maksimalno minimalno rastojanje između parova procesnih elemenata (maksimalno rastojanje koje može da se pojavi u mreži). Što je dijametar manji, mreža je bolja u pogledu povezivanja. Za linearnu mrežu, ako je n broj procesnih elemenata dijametar je d=n–1.

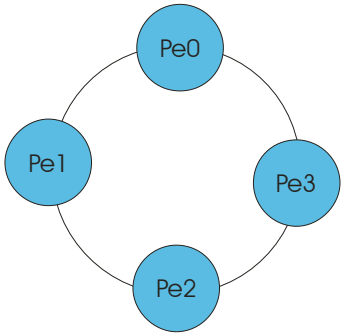
Dvodimenzionalne:

Postoji nekoliko varijanti od kojih ćemo pomenuti samo:

Prsten, Zvezda, Binarno stablo Rešetka.

Prsten:

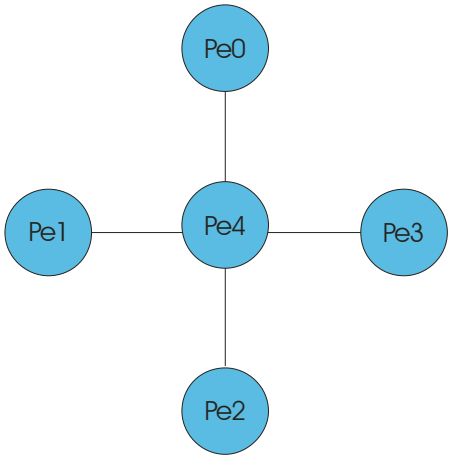
d=(n/2) *zagrade u ovom slučaju predstavljaju najveću celu vrednost razlomka.



Zvezda:

d=2 (uvek)

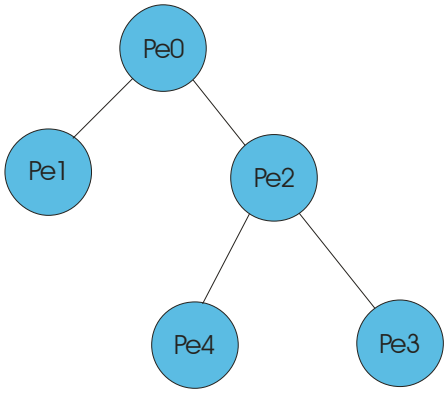
Za ovakvu konfiguraciju kritično je ako Procesni element u nekom čvoru otkaže. To je vrlo loša karakteristika sistema. U tom slučaju, zbog samo jednog procesnog elementa ceo sistem pada.



Binarno stablo:

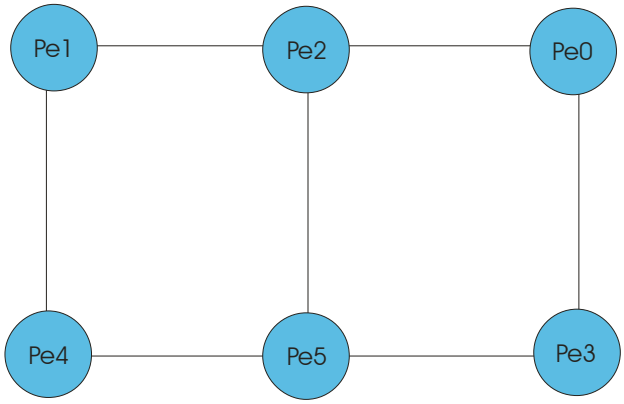
$D=2*\log_2 n$

Ovo je jako zgodna konfiguracija za aplikacije pretraživanja.



Rešetka:

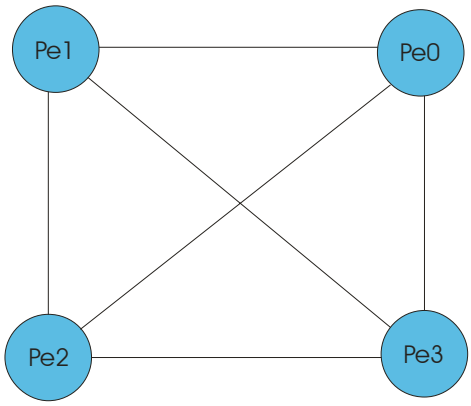
Obvo je zgodna konfiguracija za dinamičke sprežne mreže.



Trodimenzionalne:

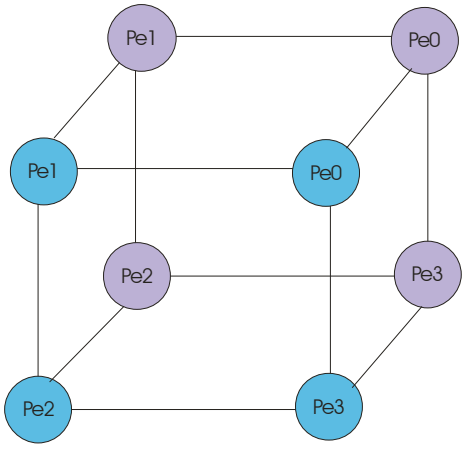
Takođe i kod trodimenzionalnih mreža postoje vazličite vrste mreža. Najčešći oblici su:

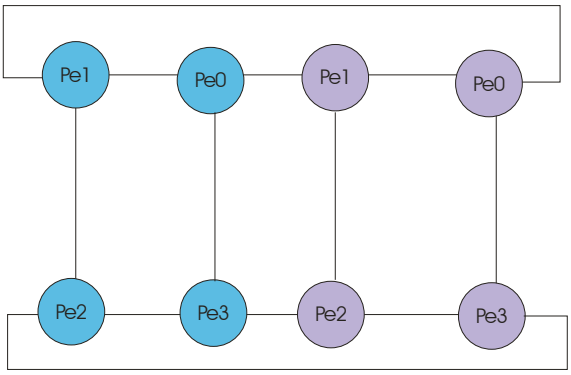
Potpuno povezana mreža kod ove mreže je naravno d=1. Loša strana ovih mreža je to što za veliki broj procesnih elemenata broj zahtevanih linija je mnogo veliki, tako da je implementacija ovakvih mreža vrlo složena. Konkretno taj broj je n*(n–1)/2. Razlog ovome je to što se svaki procesni element povezuje sa svim ostalim procesnim elementima.



Hiperkocka:

Ovo je jako popularna konfiguracija. Dobra osobina je to što ako je broj procesnih elemenata u sistemu n, onda je $D=\log_2 m$. Dakle, ukoliko je n=8 (naš slučaj) d=3. Svaki procesni element je povezan za $\log_2 n$ procesnih elemenata. Ovo bi se moglo za n=3 predstaviti i u dve dimenzije (samo za n=3 za druge vrednosti ne može).





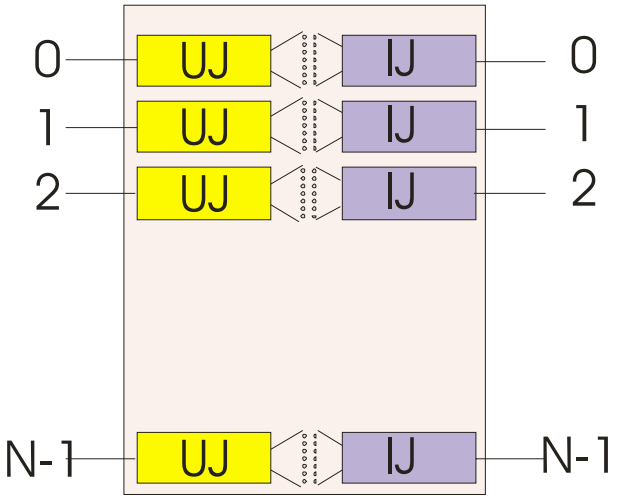
Preslikano u dvodimenzionalnu formu, hiperkocka izgledakao na slici pored same hiperkocke.

Dinamičke sprežne mreže

Dinamičke sprežne mreže se odlikuju time što veza između procesnih elemenata se uspostavlja po **zahtevu** (definisanim u programu). Dinamičke sprežne mreže sadrže **komutatore** i u odnosu na broj zahtevanih nivoa komutacionih elemenata koje koriste.

Dele se na jednostepene i višestepene.

Jednostepena dinamička sprežna mreža koja treba da poveže N procesnih elemenata sastoji se od N ulaznih jedinica i N izlaza.

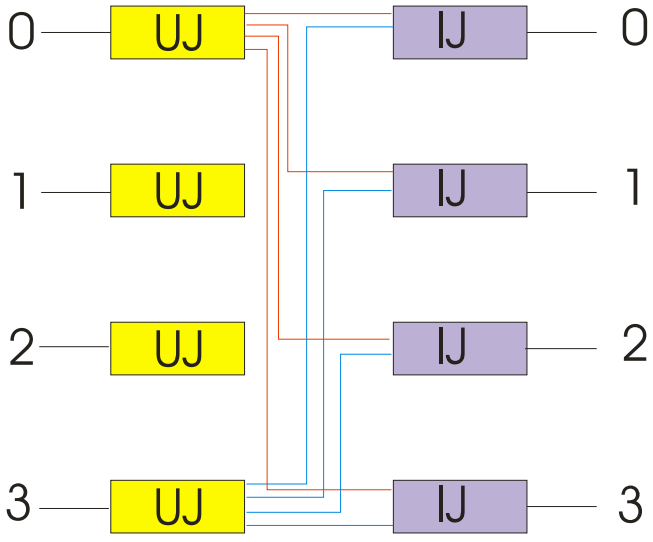


Ulazne jedinice su demultiplekseri tipa 1xD, a izlazne multiplekseri tipa Mx1. Što su veće M i D (a mora da važi da je 1<=M; d<=N) to je situacija sa stanovišta povezanosti mreže bolja. Ako je pak M=D=N imamo potpuno povezanu mrežu gde svaki procesni element može biti povezan sa bilo kojim drugim procesnim elementom.

Potpuno povezana mreža kod koje M=D=N zove se **crossbar** sprežna mreža. Kod nje imamo potpunu povezanost u odnosu na parove procesnih elemenata.

Što su M i D veći to je veća i cena mreže. Ako M i D nisu jednaki broju N, već manji, imamo sprežnu mrežu kod koje ne možemo da garantujemo da jednim prolazom može biti uspostavljena veza između proizvoljna dva procesna elementa. Ako ne može, podatak može više puta da kruži kroz mrežu da bi se ostvarilo povezivanje dva procesna elementa. Zato se ovakve sprežne mreže zovu **RE-CIRKULARNE** sprežne mreže.

Zbog složenosti ilustracije nisam crtao sve veze, jer se podrazumeva da je svaki procesni element povezan sa svim ostalim procesnim elementima.



Veze koje sprežna mreža može da ostvari definišu se preko sprežnih f-ja (ovo važi za bilo koju sprežnu mrežu). Te sprežne funkcije se matematički opisuju. Primenjuju se nad **adresom** procesnog elementa i njenom primenom se dobije adresa drugog procesnog elementa sa kojim se ostvaruje povezivanje.

Sf: X -> Y (x je adresa prvog PE a y drugog)

Dakle => $Y=S_F(x)$

Svaka sprežna mreža se definiše **skupom sprežnih funkcija** koje se mogu ostvariti (više sprežnih funkcija se koristi jer je mreža dinamička)

MESH (mreža, rešetka)

Sprežna mreža je definisana sa n sprežnih f–ja (korišćena je kod ILLIAC–a).

$S_{+1}(x)=(x+1) \bmod N$

$S_{-1}(x)=(x-1) \bmod N$

.

.

$S_{+R}(x)=(x+R) \bmod N$

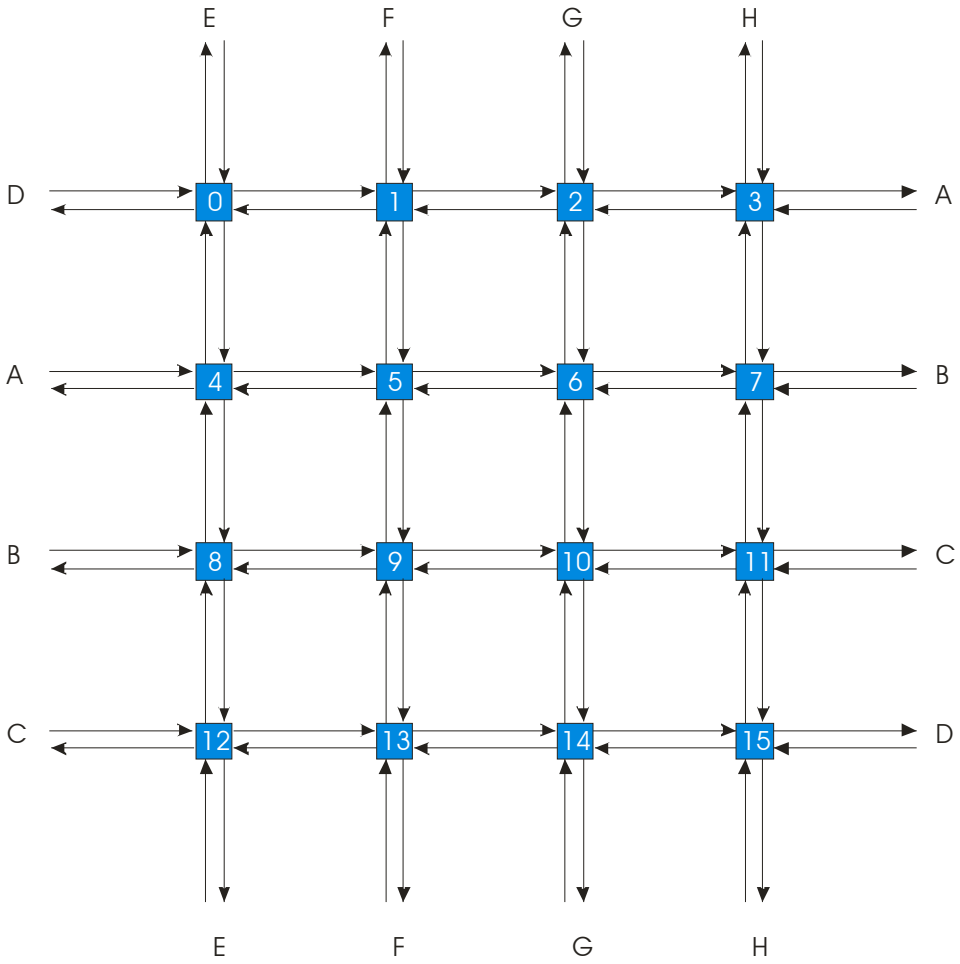
$S_{-R}(x)=(x-R) \bmod N$

Dakle,

$(-j) \bmod N = (N-j) \bmod N$

U ovako formulisanim izrazima R bi moralo da bude jednako kvadratnom korenu iz N.

Uzmimo na primer slučaj kada je N=16



Kod SIMD računara svi parovi procesnih elemenata u istom trenutku treba da izvršavaju istu sprežnu funkciju (svi aktivni procesni elementi).

Shufle/excange

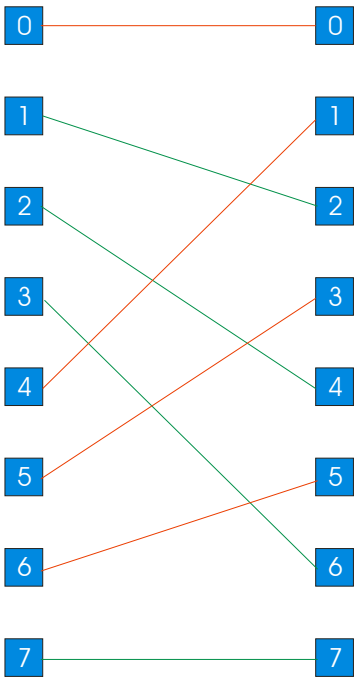
Sprežna mreža ovog tipa je definisana sa dve sprežne funkcije i to funkcijom **mešanja** (Shufle) i funkcijom **zamene** (Excange).

$S(B_{m-1}, b_{m-2}, \ldots\ldots b_1, b_0) = b_{m-2} \ldots b_1 b_0 b_{m-1}$

$E(b_{m-1}, b_{m-2}, \ldots\ldots b_1, b_0) = b_{m-1} \ldots\ldots b_1 \; b_0{}^*$

Zvezdica u ovom izrazu govori da se radi o komplementu!

Sa ovim Bx obeležena je adresa procesnog elementa u binarnom obliku.



Ukoliko bi za povezivanje koristili S funkciju, ono je ekvivalentno savršenom mešanju špila karata.

Dakle,

element 0 se povezuje sa elementom 0

element 4 se povezuje sa elementom 1

element 1 se povezuje sa elementom 2

element 5 se povezuje sa elementom 3

element 2 se povezuje sa elementom 4

element 6 se povezuje sa elementom 5

element 3 se povezuje sa elementom 6

element 7 se povezuje sa elementom 7

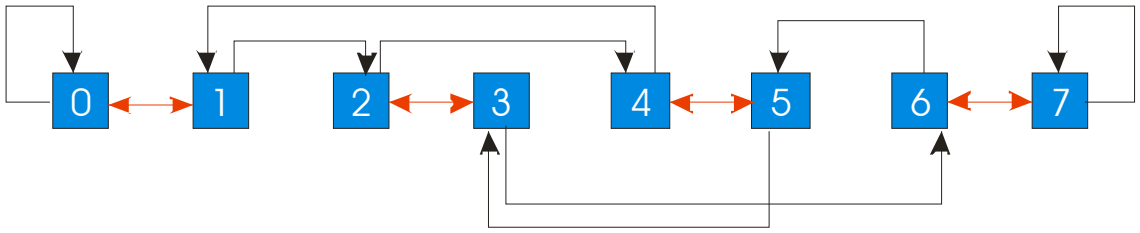
Dakle, zaključujemo da su nule (000) i sedmice (111) ostali nepovezani. Tj. oni ne komuniciraju ni sa kim.

Zbog toga se uvodi excange funkcija koja omogućava da i oni komuniciraju.

E(000) –> 001

E(111) –> 110

Šematski bi to izgledalo ovako:



Crvenm bojom su prikazane E(excange) veze, a crnom S(shufle) veze.

Kub sprežne mreže

Ova mreža je definisana sa $m=\log_2 N$ sprežnih funkcija. Te funkcije označene su sa C_i.

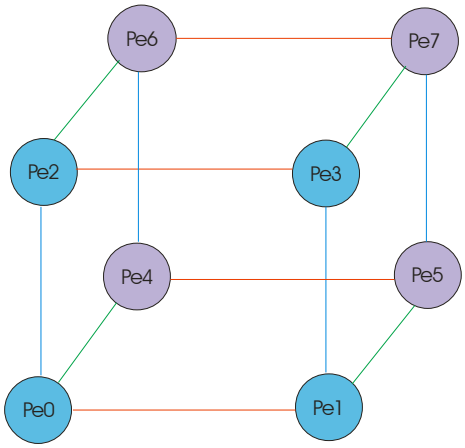
$C_i(b_{m-1},b_{m-2},.....b_{i+1},b_i,b_{i-1}.....b_1,b_0) = b_{m-1},....b_{i+1}\ b_i^*,\ b_{i-1},...b_0$

Uzmimo na primer da je N=8. tada bi postojale tri sprežne funkcije:

C_0: 0–1,2–3,4–5,6–7

C_1: 0–2,1–3,4–6,5–7

C_2: 0–4,1–5,2–6,3–7



različitim bojama prikazane su različite sprežne funkcije.

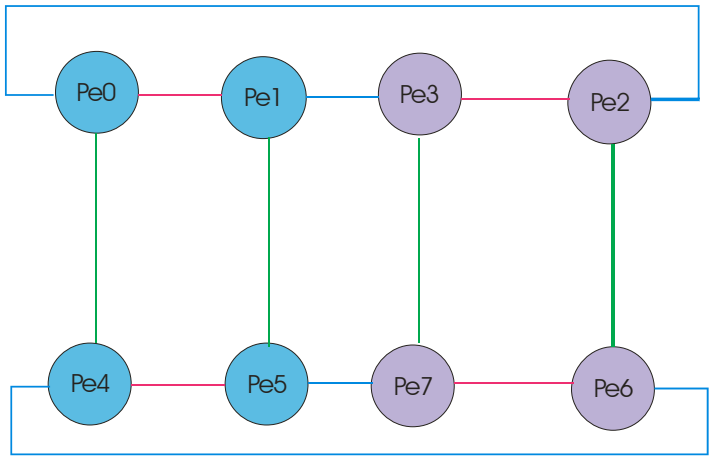
NPR:

crvenom bojom je prikazana funkcja C0

Plavom bojom je prikazana funkcija C1

A zelenom bojom je obeležena sprežna funkcija c2

Ovu sprežnu funkciju možemo predstaviti u ravni na sledeći način:


Pe0
Pe1
Pe3
Pe2
Pe4
Pe5
Pe7
Pe6

Primer:

Računanje vrednosti polinoma:

$$P_n(x) = \sum_{i=0}^{n-1} a_i x^i = a_0 + a_1 x + ... + a_{n-1} x^{n-1}$$

Neka je PE_i a_i i x_i u početnom trenutku. U PE se pamti i vrednost maske.

Ako imamo n procesora problem može da se reši u $2\log_2 n$ koraka pri čemu imamo dve oktivnosti:

- Izračunavanje članova i to se radi u $\log_2 n$ koraka
- Sumiranje

Na kraju će svaki PE sadržati vrednost (celu) polinoma.

ALGORITAM:

If m=1 then $a_i = a_i * x$

Endif

X=x*x

Shufle(m_i)

U tri ovakva koraka dobijamo sabirke. U sledeća tri koraka se obavlja sumiranje...

Višestepene sprežne mreže

Višestepene sprežne mreže sastoje se od **komutacionih elemenata** koji čine jadan stepen mreže. Veze između stepena su fiksne. One mogu biti različitog oblika: leptir, savršenog mešanja itd. Ono što karakteriše višestepene sprežne mreže je izgled komutacionog elementa tj. način povezivanja susednih elemenata i način upravljanja.

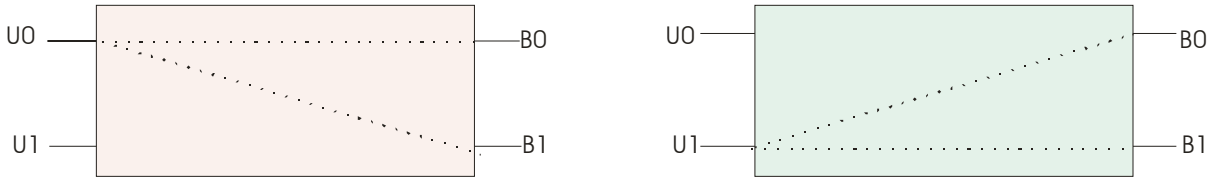
Komutacioni element u opštem slučaju može imati a ulaza i b izlaza, pri čemu nije obavezno da je a=b. Međutim u praksi imamo slučaj da je $a=b=2^k$.

Cena komutacionog elementa raste sa brojem ulaza i brojem izlaza, tako da se najčešće koriste komutacioni elementi sa a=b=2.

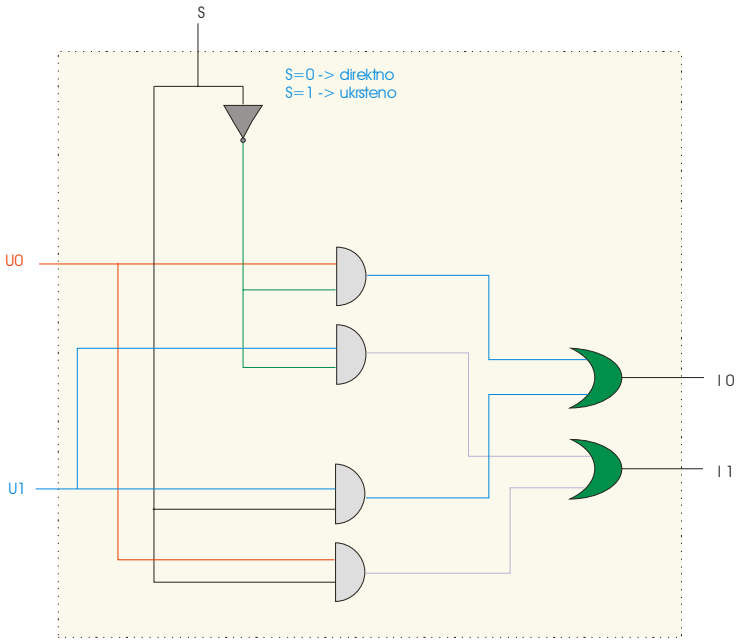
Komutacioni element koji ima a=b=2 može u zavisnosti od načina sprezanja biti **dvofunkcijski** i **četvorofunkcijski**.

Kod dvofunkcijskog komutacionog elementa moguće je direktno povezivanje i ukršteno povezivanje.

Kod četvorofunkcijskih elemenata pored ova dva tipa načina povezivanja moguća je i gornja i donja emisija. Kao što je to prikazana na donjoj slici.



Najjednostavniji komutacioni element je 2x2 On je **dvofunkcijski** i ne sadrži nikakvu memoriju. Sastoji se samo od I ili ILI kola. On je prikazan na sledećoj slici:



Ako višestepena mreža treba da poveže N procesnih elemenata ona ima $m=\log_2 N$ stepena, a svaki stepen ima $N/2^k$ komutacionih elemenata (naravno, ako svaki ima 2^k ulaza).

Aktivni mrežni elementi su **komutacioni elementi**. Veze između susednih stepena tj. komutacionih elemenata su fiksne, a sami komutacioni elementi mogu realizovati različite f–je.

Po načinu upravljanja delimo ih na:

Nezavisno upravljanje stepenima mreže:

U ovom slučaju se jedinstveni signal koristi da sve komutacione elemente jednog stepena postavi u isti položaj. Potrebno je m upravljačkih signala za upravljanje.

$N=2^m$

$M=\log_2 N$

Nezavisno upravljanje komutacionih elemenata:

Ovde postoji poseban upravljački signal za svaki komutacioni element. Broj zahtevanih linija je;

M x N/2 = broj zahtevanih upravljačkih linija

M broj stepena, N/2 komutacionih elemenata

Nezavisno upravljanje na nivou blokova u okviru jednog stepena:

Jednim prolaskom kroz mrežu, može se ostvariti povezivanje proizvoljnog para procesnih elemenata.

Generalizovani kub

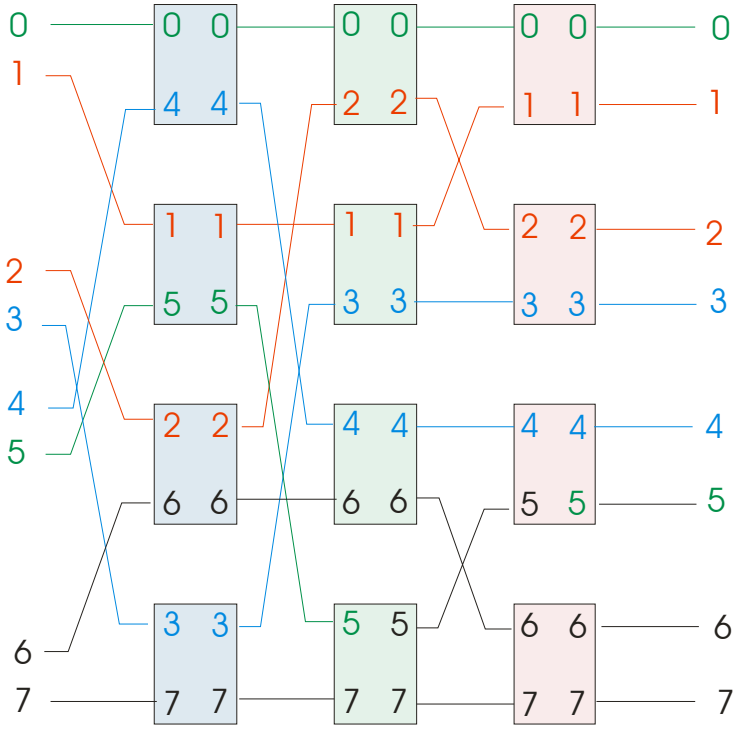
Ova sprežna mreža se sastoji od $N=\log_2 N$ **stepena** i sa N/2 komutacionog elementa tipa 2u2 (čita se dva u dva) u svakom elementu. Komutacioni elementi su **četvorofunkcijski**.

Svaki stepen obavlja jednu sprežnu funkciju kub–sprežne mreže.

$C_i(b_{m-1},....b_i,.....b_0) = b_{m-1},.....b_i,.....b_0$

U ovom izrzu i se može menjati od 0 do m–1

Na primer: uzmimo slučaj da je N=8. Dakle, 3 stepena sa po 4 komutaciona elementa.



Između svakog para postoji jedan **jedinstveni** put. Ovo nije dobro sa stanovišta **pouzdanosti** mreže. Nema alternativnih puteva Tj. ako se desi greška na jednom putu, veza se ne može uspostaviti jer nema alternativnih puteva. U mreži mogu da nastupe konflikti, kada dva para zahtevaju korišćenje istih komunikacionih kanala. Ako ne postojie baferi, prosto se jedan zahtev odbacuje, a ako bafer postoji jedan zahtev se baferuje.

Definisanje puta

Da bi se definisao put poruke tj. put koji poruka prolazi da bi od jednog došla do drugog elementa, svakoj poruci je potrebno ugraditi m–bitno **zaglavlje**. Svaki bit treba da definiše u koji položaj treba da se postavi komutacioni element u odgovarajućem stepenu.

Postoje dve vrste komunikacije:

- Jedan na jedan
- Emisija (jedan na više)

Jedan na jedan:

Što se tiče druge varijante moguća su dva načina za formiranje zaglavlja kojima se definiše put poruka kroz mrežu.

- XOR (od adrese izvora do adrese odredišta)
- Na osnovu odredišta adresa

Što se tiče **XOR varijante**, uzmimo na primer S=Sm–1,...Si....S1,S0 i D=dm–1,....di...d1,d0. Zaglavlje koje se dodaje poruci imaće sledeći oblik.

$$T = S \oplus D = t_{m-1}.....t_i.....t_1, t_0$$

Ova poruka putovaće od **Source** do **Destination** mesta.

Da bi se definisao put komutacioni element treba da ispita u i–tom stepenu i–ti bit zaglavlja. Ukoliko je taj bit **0**, onda će se koristiti **direktno povezivanje**, a ukoliko je **1** onda će se koristiti **ukršteno povezivanje**. Razlog za ovo se krije u strukturi sprežne mreže i definiciji sprežnih funkcija.

$T_i = S_i \oplus D_i = 1$ ako se razlikuju adresa izvora i adresa odredišta, sprežna funkcija Ci mora da se implementira.

$T_i = S_i \oplus D_i = 0$ ako se ne razlikuju adrese izvorišta i odredišta, onda nije potrebno realizovati Ci

Dobra osobina je to što odredište na osnovu poruke sa pristiglim zaglavljem može da utvrdi od koga poruka dolazi.

Što se pak tiče **varijante na osnovu adrese**, na osnovu adrese odredišta se definiše na koji izlaz se prosleđuje pristigla poruka.

Ukoliko je ona **0**, onda se prosleđuje na **gornji izraz** komutacionog elementa a ako je **1** onda se prosleđuje na **donji izlaz**. Ovde se razlog krije u strukturi sprežne mreže. komutacioni element u stepenu i na gornjem izlazu u i–toj poziciji uvek ima 0, a na donjem izlazu je uvek jedinica.

Dobra osobina je to što odredište na osnovu pristigle poruke sa zaglavljem može da utvrdi da li poruka stiže po korektnom mrežnom izlazu.

Emisija (jedan na više–emitovanje)

U ovom slučaju jedan izvor šalje u 2^l odredišta. Dakle, možemo govoriti i o emisiji jednog izvora u dva odredišta. Za definisanje puta se koristi **varijanta sa XOR zaglavljima**. Da bi prva emisija bila moguća, mora da bude zadovoljeno:

- Da broj odredišta bude stepen dvojke
- Da adrese odredišta smeju da se razlikuju samo na jednoj bitskoj poziciji (**Hemingovo rastojanje** je 1)

Na primer:

S=5 tj. 101

E=4 tj. 100

F=6 tj. 110

$$T_E = 101 \oplus 100 = 001$$
$$T_F = 101 \oplus 110 = 011$$

Jedan zahteva **direktno**, a drugi **ukršteno** povezivanje, zato dolazi do emisije. Da bi za slučaj emisije mogli da definišemo put, moramo da znamo u kojim stepenima dolazi do emisije. U slučaju emisije zaglavlje se sastoji iz 2 dela:

R = Prvi deo je m–bitan i sadrži informaciju o putu kao kad nema emisije

B = Drugi deo sadrži informaciju o tački grananja.

Za R možemo da uzmemo bilo koje od individualnih zaglavlja ($R=T_E$ ili $R=T_F$)

Što se tiče B dela, do emisije dolazi tamo gde se adrese međusobno razlikuju.

$$B = T_E \oplus T_F = E \oplus F$$

Dakle, prvih m bitova važi kada nema emisije, a drugih m važe za tačke u kojima dolazi do grananja. Dakle, celo zaglavlje je dužine 2m.

Kad pristigne zaglavlje za i–ti stepen ispituje se Ri i Bi tj. i–ti bit R i B dela. Da bi se utvrdilo u koj položaj se postavlja komutacioni elemnt prvo se ispituje Bi. Ako je Bi jednko **jedinici**, onda se radi o emisiji, a ako je Bi jednako nuli, onda se nastavlja sa **čitanjem** Ri.

Dakle, ako je **Bi=1** Ri se uopšte ne ispituje. **Vrsta emisije** (gornja ili donja) se određuje u zavisnosti od toga po kom je ulazu stigla poruka.

Ako je pak **Bi=0** u zavisnosti od Ri komutacioni element se postavlja u odgovarajući položaj. Tj. Ukoliko je Ri jednako **jedinici** postavlja se u ukršeni položaj (ukršta) a ukoliko je Ri jednako **nuli** onda se postavlja u **direktan** položaj (direktno provodi).

Na primer:

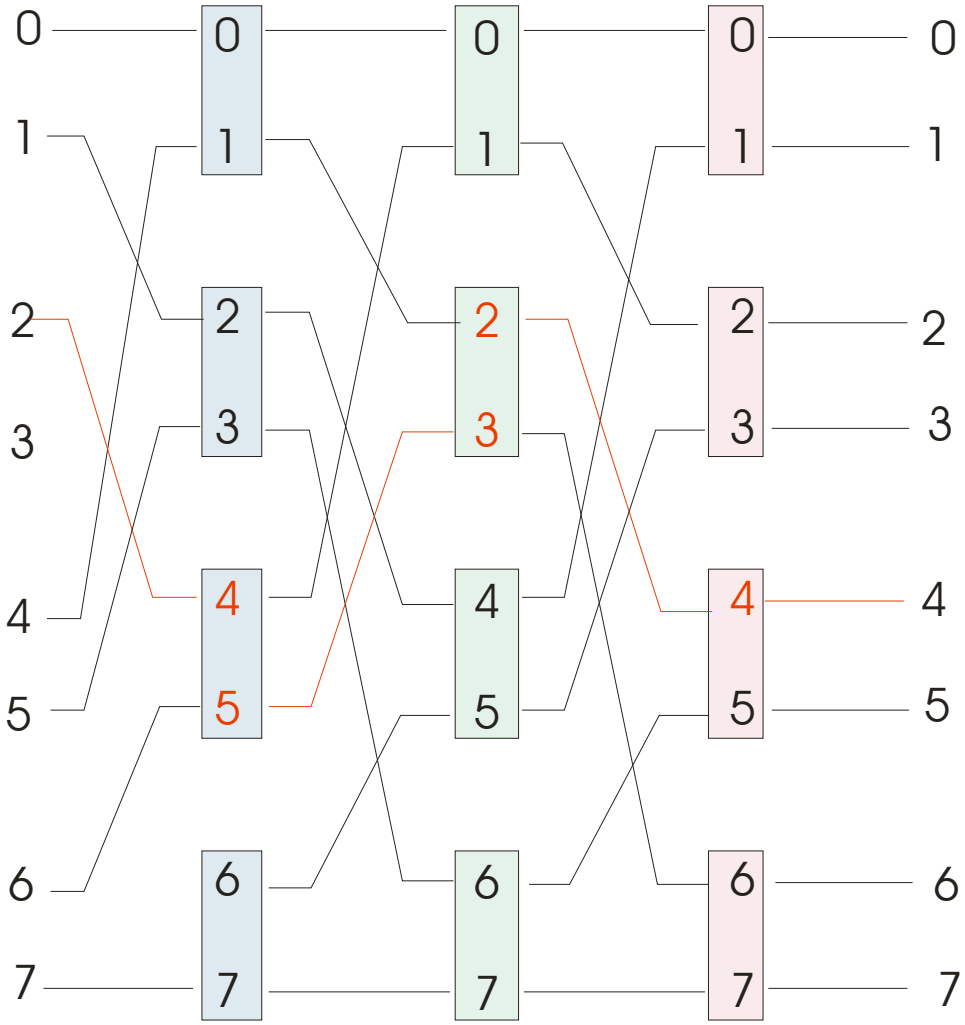
Zaglavlje je (001, 010). Iz ovoga vidimo da je T_E = 001 a da je (T_E XOR T_F)=010

Da bi emisija bila moguća, broj odredišta mora biti neki stepen dvojke a odredišne adrese mogu da se razlikuju za najviše J binarnih pozicija dok se na ostalih n–j binarnih pozicija one moraju slagati.

Omega sprežne mreže

Omega sprežna mreža obavlja sprežne funkcije mešanja i zamene. Za $N=2^m$ mreža ima m stepena (elemenata) sa N/2 komutacionih elemenata u svakom stepenu. Komutacioni elementi obavljaju funkciju zamene, dok su veze između stepena tipa mešanje.

Za N=8 m=3 sledi slika:



Za **rutiranje** se koristi odredišna adresa u zaglavlju. Po istom principu, kao kod kub sprežne mreže definiše se na koj se izlaz sprežne mreže prosleđuje prispela poruka.

$D=d_{m-1},....d_i....d_0$

Ako je di jenako **jedinici**, onda se vrši po donjem a ako je jednako **nuli** onda se vrši po gornjem. Ako nastupi greška u bilo kom komutacionom elementu ali ne bilo kom komunikacionom kanalu veze između odgovarajućeg para procesnih elemenata više nikada neće moći da se uspostave. Da bi se poboljšala otpornost mreža na greške moguće je dodati još hardverskih stepena.

Na primeru generalizovoanog kuba pokazano je kako se povećava otpornost na greške (napravićemo da je otporan na jednostruke greške). Ekstra stepeni se dodaju na ulaz mreže. Tako se dobija **ESC**–ExtraStageCube mreža.

ESC–ExtraStageCube

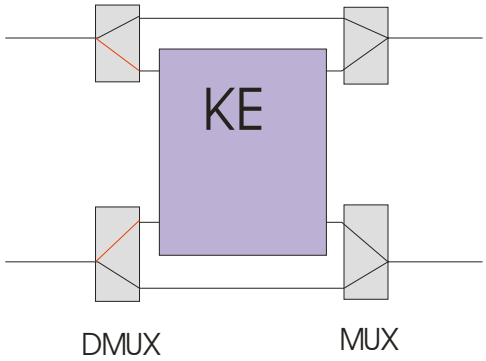
Stepen koji se dodaje na ulaz mreže obavlja funkciju C_0:

$C_i\ (b_{m-1}.....b_i....b_0)=(b_{m-1}....\bar{b}_i.....b_0)$ $\quad i=0,.....m-1$

Srvenom bojom je označena negacija bita tj. ona je ekvivalentna oznaci >**nadvučeno**<!!

Sada u mreži postoje dva stepena koji obavljaju istu funkciju. Pored dodavanja ekstra treba svakom ulazno izlaznom stepenu dodati hardver koji će omogućiti premošćavanje ekstra stepena (taj hardver se obično sastiji od multipleksera i demultipleksera).

Hardver koji omogućava premošćavanje:



Svi KE (komutacioni elementi) na ulazu i na izlazu imaju ovaj hardver. Ovo omogućava da između svakog izvorišnog i odredišnog stepna postoje po dva puta, pa ako se na primarnom putu javi greška, možemo preći na sekundarni put.

Postavljanje stepena m i stepena 0 obavlja upravljačka jedinica. Pri tome svi komutacioni elementi su postavljeni u isti položaj (aktivan ili pasivan). Normalno je mreža konfigurisana tako da je S0 aktivan a Sm premošćen (i to se svodi na prvobitnu mrežu tj. generalizovani kub. Ova mreža može da podnese otkaz jednog komutaionog elementa tj. tolerantna je na jednostruke greške. Te greške mogu da nastupe u komutacionim elementima ili u komunikacionom kanalu, ali nikako nesmeju nastati u U/I portovima mreže.

Dakle, ova mreža može da radi čak i ako postoji 1 greška.

- Prvo se obavlja **detekcija greške**. Mreža se može testirati tako što se testira za poznate vrednosti ulaza i izlaza. Ako se utvrdi neslaganje, onda zaključujemo da negde postoji greška.
- **Lokalizacija greške**. Da bis e jednostruka greška lokalizovala potrebno je izvršiti 12 različitih testiranja koja lokalizuju grešku.
- **Rekonfiguracija mreže**.
    - Ako je greška u stepenu m, ništa se ne preuzima jer tada je taj stepen premošćen i sve radi normalno.
    - Ako je greška u stepenu 0. Tada se mreža postvlja tako da Komutacioni elementi u stepenu m budu aktivni, a komutacioni elementi u S0 se premošćavaju.
    - Ako je greška negde između, Si onda se i stepen m i stepen 0 postavljaju u aktivno stanje. Ako su aktivni komutacioni elementi u Sm i u S0, tada između bilo kod para

procesnih elemenata postoje dva puta kroz mrežu koja ne koriste zajedničke komutacione elemente u stepenima m–1,...,1. to znaĐči da ako je nastupila greška na jednom putu, poruka nesmetano dolazi preko drugog puta. Označimo zaglavlja za ove puteve sa Tp (p–primarni, zaglavlje kao kod normalnog kuba sa premošćavanjem Sm), Ts(s – sekundarni, odgovara slučaju kada su aktivni ijedan i drugi).

Sekundarni put koristi postavljanje komutacionih elemenata u stepenu S0 u **položaj ukršteno** (obavezno obavlja f–ju C_0),a primarni u **položaj diekno** (kao da se Sm premošćen).

Pod određenim uslovima može se detektovati i dvostruka greška, ali ne uvek.

Najveći broj grešaka su upravo jednostruke greške (99%) pa je zato dovoljno ostvariti postojanost na jednostruke greške.

Sve u svemu tehnike za detekciju grešaka se mogu podeliti na **algoritamske tehnike** i tehnike koje koriste **kontrolisanu redundansu** (hardverski ili softverski).

MIMD RAČUNARI

MIMD računari predstavljaju najopštiju i najmoćniju klasu paralelnih računara. MIMD računar se sastoji od 2 ili više procesora približno jednake moći izračunavanja. U MIMD sistemu ne postoji centralna **Control Unit** i umesto nje svaki procesor poseduje svoju upravljačku jedinicu i logičku jedinicu. Dakle, radom procesora upravlja njegova sopstvena upravljačka jedinica.

U istom trenutku izračunavaju se različite instrukcije nad različitim podacima. Te instrukcije predstavljaju deo nekog zajedničkog problema koji treba da se reši. Svaki procesor izvršava određeni skup **taskova** (procesa).

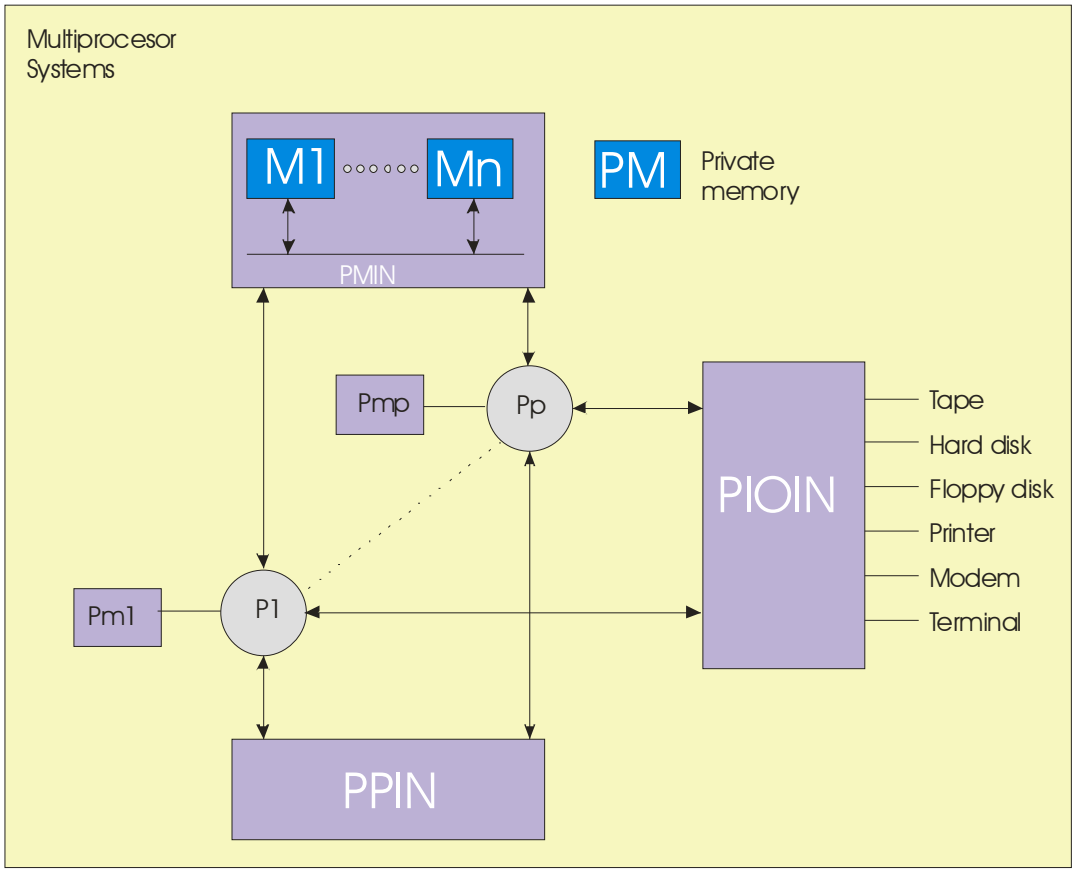
Logično se nameće potreba za mehanizmima za **komunikaciju** između procesa koji se odvijaju u različitim procesorima. Komunikacija između procesora se može ostvariti preko **deljive memorije** ili preko **zajedničke memorije** u kojoj se mogu smeštati deljivi podaci. Druga alternativa je komunikacija među procesima putem **slanja poruka**.

Neophodno je obezbediti i sredstva za sinhronizaciju procesa. **Tehnike** i **sredstva** za komunikciju mogu se koristiti i za **sinhronizaciju**, samo što se u ovom drugom slučaju umesto podataka razmenjuju **upravljačke promenljive**. Sinhronizacijom se obezbeđuje korektan red izvršenja zadataka i uzajamno isključivo pravo pristupa deljivim resursima.

U odnosu na to kako se obavljaju komunikacije između procesa, sisteme delimo na:

- Čvrsto spregnute (multiprocesori)
- Slabo spregnute (multiračunare)

Multiprocesorski sistemi

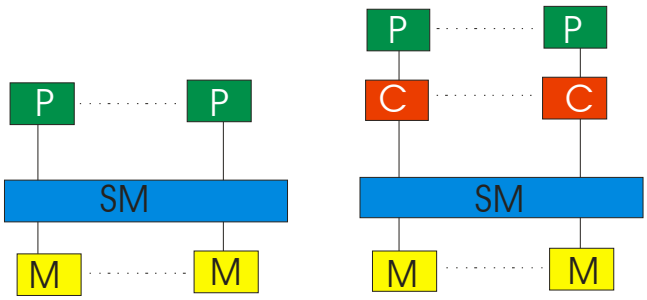


Multiprocesorski sistemi se sastoje od dva ili više homogenih računara. Svi procesori dele pristup **zajedničkoj memoriji**. Ta memorija je označena sa M1.....Mn. Pristup se obavlja preko **sprežne mreže PMIN**. **Čvrsta sprega** se odnosi na postojanje istog adresnog prostora. Svaki procesor može da ima i malu privatnu memoriju koja je označena sa **PM**. Sinhronizacija između procesora se može ostvariti i preko sprežne mreže **PPIN**. Svi procesori dele pristup zajedničkim U/I uređajuma – **PIOIN**.

Ceo sistem može biti organizovan korišćenjem samo jedne sprežne mreže. Sprežne mreže su **dinamičkog tipa** tj. veza se ostvaruje dinamički (po zahtevu). Najjednostavnija sprežna mreža je **zajednička magistrala**. Cena je najniža, reda je O(p) ali je i propusnost mreže najniža.

Drugo rešenje je korišćenje **crossbar** sprežne mreže koja poseduje potpunu povezanost procesora i magistrale, ali loša strane je to što je neekonomična. cena je reda $O(P^2)$.

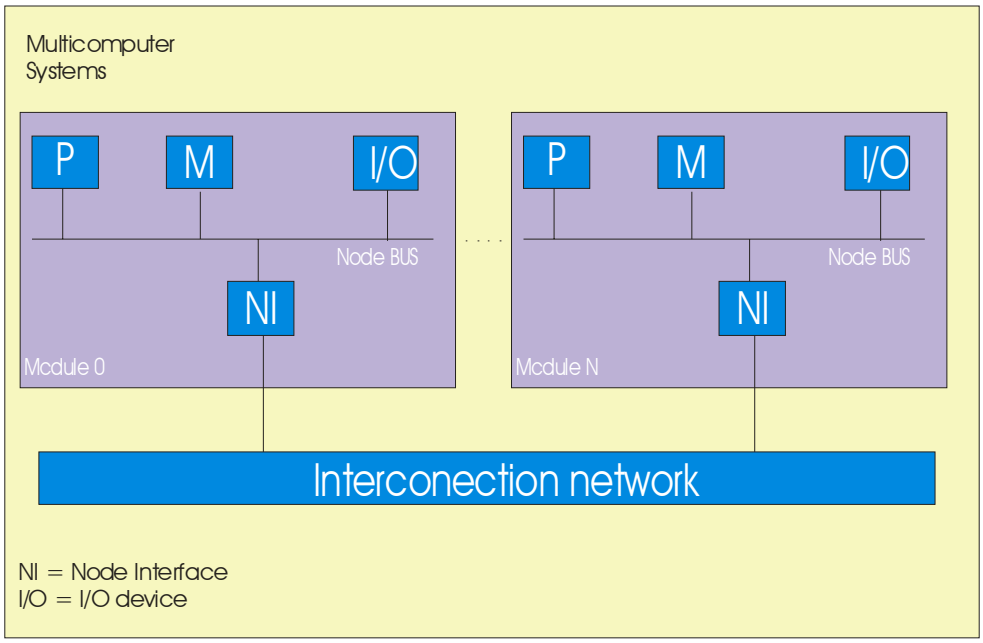
Kao kompromis se nameće korišćenje jednostepenih i višestepenih sprežnih mreža. bez obzira o kakvoj se mrežnoj mreži radi, može doći do konflikta (zato što se zahteva ili korišćenje istog puta ili memorijskog modula). Broj tih konflikata i vreme kašnjenja kroz speržnu mrežu se može smanjiti uvođenjem privatnih keš memorija koje se postavljaju između procesora i sprežne mreže.



Javlja se problem! Može se desiti da u jednom trenutku više procesora jednovremeno sadrži kopiju istog bloka podataka iz glavne memorije i ako dođe do lokalne modifikacije jednog bloka, dolazi do neusaglašenosti tj. **nekonzistentnosti**.

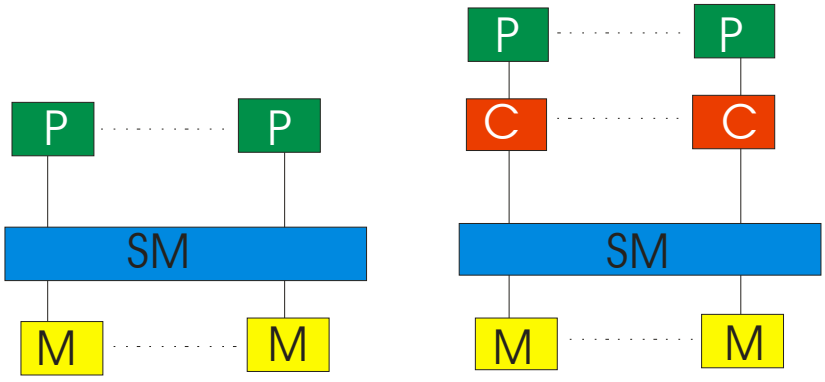
Multiračunarski sistemi

Oni nemaju zajednički adresni prostor. Svaki procesor ima svoju privatnu memoriju kojoj ne može da pristupi drugi procesor. Razmena podataka se vrši eksplicitnim **slanjem poruka**. Multiprocesori podsećaju na računarske mreže.



Sprega sa drugim modulima se ostvaruje preko čvornog interfejsa NI (node interface). Ne zahteva se da svi procesori budu istog tipa.

Moderni računari koriste **hardverski ruter**, tako što svaki modul (računarski čvor) povezan na jedan ruter, a ruteri su dalje povezani između sebe i da bi se ostvarila komunikacija između dva čvora potrebno je ostvariti vezu između rutera.



Sprežne mreže su **statičkog tipa**, a najčešće korišćene topologije su: Prsten, tablo, hiperkub itd

Radom svakog multiračunara upravlja jedinstveni operativni sistem. Otkaz jednog računara u mreži je vidljiv za korisnika. Otkaz jednog procesora u multiračunarskom sistemu može da degradira performanse ali sistem je i dalje operativan.

Zajednička magistrala

Odnosi se na multiprocesore. Najjednostavniji sprežni put. Na taj sprežni put povezani su svi uređaji sistema. Ovakva sprežna mreža je **pasivna**. Operacijom prenosa po magistrali u potpunosti upravlja interfejs izvorišne i odredišne jedinice. Jedinica koja želi da uspostavi prenos mora da **ispita dostupnost** magistrale, pa onda **adresira odredište** i **proverava** da li je ono spremno za komunikaciju i ako jeste **kreće prenos**, a nakon toga magistrala se oslobađa.

Ploče koje se povezuju na magistralu mogu biti **gospodar** i **sluga** (master i slave). **Gospodar** je ploča koja može inicijalizovati zahtev za korišćenje magistrale i da oni diktiraju prenos, a **sluge** mogu samo da primaju podatke. Moguća je i ploča kombinovanog tipa ali u jednom trenutku ploča (uređaj je ili jedno ili drugo. Pošto se na zajedničku magistralu povezuje veći broj potencijalnih gospodara, a magistrali može biti dodeljen samo jedan gospodar, treba obezbediti **arbitražu** da ne bi dolazilo do konflikata pri dodeli magistrale. Arbitraža se može izvesti kao **statička** i kao **dinamička**.

Kod **statičke** arbitraže raspored korišćenja magistrale je unapred definisan (najčešće je kružni) na primer sekvenca uređaje kruži ABCD ABCD ABCD ABCD i tako dalje. U svakom treutku je jedan od uređaja gospodar a u sledećem trenutku je gospodar onaj uređaj iza njega. Ovaj način arbitraže je jednostavan ali ne valja što se ne vodi računa o stvarnim potrebama potencijalnog gospodara (može se desiti da uređaj dobije magistralu čak i ako mu ne treba).

Kod **dinamičke** organizacije arbitraže se korišćenje magistrale određuje po zahtevu. Dodela i oslobađanje se ostvaruju određenim politikama dodele. **Politike dodele** magistrale mogu biti zasnovane na **prioritetima** (tada svaki potencijalni gospodar ima svoj nivo prioriteta i u slučaju više zahteva, magistrala se dodeljuje uređaju sa najvišim prioritetom.

Druga tehnuika je zasnovana na **nepristrasnosti**. Svakom gospodaru koji je uputio zahtev za pristup magistrali mora se dodeliti magistrala pre nego što se nekom drugom dodeli po drugi put magistrala. Ovim se sprečava mogućnost da brzi uređaji zaguše sporije uređaje.

Naravno, postoji i **kombinovana varijanta**. Na primer, ulazno–izlaznim uređajuma se dodeljuje po principu prioriteta, a procesnim elementima po principu nepristrasnosti.

Kao što postoje politike za dobijanje magistrale, tako postoje i politike za oslobađanje magistrale.

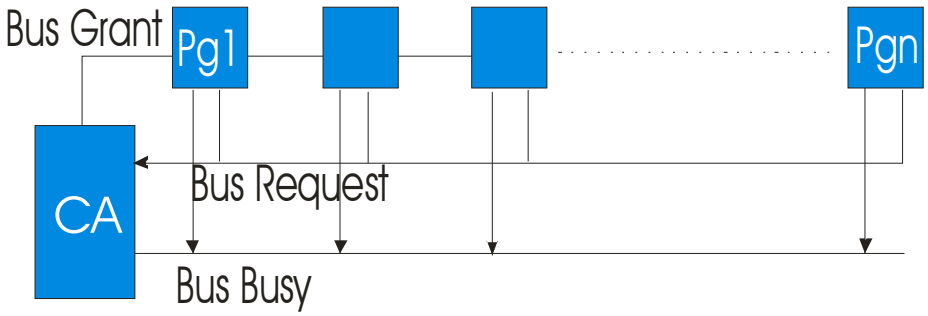
- **Oslobađanje po zahtevu** (čak i kada se transakcija završi gospodar ne oslobađa magistralu dok ona ne postane potrebana nekom drugom uređaju
- **Oslobađanje nakon obavljene transakcije**
- **Istiskivanje** tj. gospodar sa višim prioritetom zahteva od gospodara sa nižim prioritetom da mu oslobodi magistralu iako nije završio sa prenosom

Hardver za arbitražu na magistrali može biti realizovan kao **centralizovan** hardver ili **decentralizovan** hardver.

Centralizovana arbitraža: Hardver za arbitražu je koncentrisan na jednom mestu, na primer, u nekom modulu postavljenom na magistralu ili čak može biti izveden kao nezavisni hardverski modul (**centalni arbitar**). Svi zahtevi za dodelu magistrale se upućuju ovom centralnom arbitru, a on na osnovu usvojene politike dodele vrši dodelu magistrale jednom potencijalnom gospodaru. Hardverski mehanizam koji se koristi za dodelu magistrale mogu se realizovati kao:

- Lančani zahtevi i zahtevanje
- Nezavisni zahtevi i zahtevanje

Deljivi zahtevi i lančano zahtevanje:

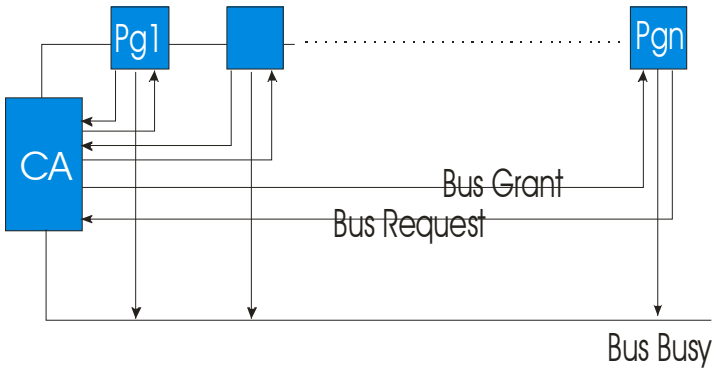


Linija **Bus Grant**: Svi potencijalni gospodari su povezani u lancu. Ovo vezivanje ima neke dobre i neke loše strane.

Dobro: Mali broj linija za arbitražu.

Loše: Fiksni prioriteti pri čemu najviši prioritet imaju oni elementi koji su bliži centralnom arbitru. Ovo može dovesti do toga da uređaji na kraju >>**umiru**<< od gladi za magistralom što se često ispostavi kao usko grlo sistema.

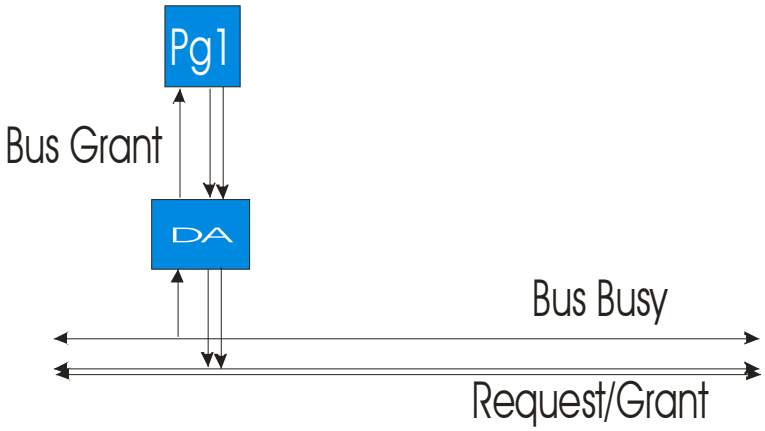
Nezavisni zahtevi i zahtevanje:



Centralni arbitar na osnovu politike dodele vrši dodelu magistrale. Savaki element ima svoje linije (nisu ulančani). Ovo je brža varijanta od malopre pomenute ali je i skuplja. Skupoća se ogleda u broju potrebnih linija.

Distribuirana arbitraža:

Svaki potencijalni gospodar ima svoj **distribuirani arbitar** sa jedinstvenim arbitražnim brojem koji se koristi za razrešenje konflikta.



Kada potencijalni gospodar traži magistralu, on prosleđuje AB do DA (**distribuiranog arbitra**). On zatim prosleđuje taj AB na linije Request/Grant. Od svih tih zahteva formira se **zbirni arbitracioni broj #AB** i svaki distribuirani arbitar poredi taj broj sa svojim AB i to počev od bita najveće težine. Ako ustanovi da je njegov **AB** manji od zbirnog on povlači svoj zahtev. Na kraju kao pobednik ostaje onaj potencijalni gospodar koji je imao **najveći AB**.

Kada **DA** prepozna njegov broj, one linije (Request/Grant) postaju **Grant linije** i time je magistrala dodeljena tom potencijalnom gospodaru.

U realnim istemamima za nadgledanje magistrale koriste se sledeće tehnike:

- Multy Bus
- Future Bus

Sinhronizacija i komunikacija procesa u MIMD sistemima

Osnovna jedinica konkurentnog izvrsenja u MIMD sistemu je **proces**. To je deo programa (niz naredbi) koji se izvršava sekvencijalno na jednom procesoru. Veoma retko su procesi koji se izvšavaju paralelno potpuno nezavisni, tj. procesi zahtevaju nekakvu razmenu podataka ili medjusobnu **komunikaciju i sinhronizaciju** (zbog toga je korektnije reći **konkurentni** umesto **paralelni** procesi). Zbog toga MIMD sistem mora sadržati sredstva koja podržavaju interprocesorsku komunikaciju i sinhronizaciju.

Fundamentalni postulat koji se odnosi na interprocesnu komunikaciju MIMD sistema je da se ne mogu uvoditi nikakve pretpostsvke o relativnoj brzini izvršenja procesa. Ovo znaci da aka dva procesa P1 i P2 treba da komuniciraju oni moraju biti korektno sinhronizovani da bi komunikacija mogla da se obavi. Dalje, ova sinhronizacija mora biti ugradjena u sam proces, tj. mora biti programirana.

Tradicionalno su mehanizmi interprocesne komunikacije bili zasnovani na korišćenju deljivih promenljivih. Ovaj prilaz je i dalje dominantan kod **multiprocesorskih** sistema, tj. sistema sa deljivom memorijom. Altemativni prilaz je koriscenje tehnike **slanja poruka**. Ovaj prilaz je postao atraktivan sa pojavom jezika kao sto su CSP, OCCAM, Ada. Na arhitekturnom nivou, mehanizmi sinhronizacije i komunikacije putem razmene poruka preovlađuju kod sistema sa distribuiranom memorijom (tj. kod multiračunarskih sistema).

Sinhronizacija i komunikacija procesa kod multiprocesorskih sistema

Posmatrajmo dva procesa P1 i P2 koji se izvršavaju na dva različita procesora paralelno. U nekom trenutku svog izršenja oba procesa zahtevaju da modifikuju zajedničku promenljivu D koja je zapamćena u glavnoj memoriji. Neka ti delovi programa izgledaju ovako:

```
PI: .......
LOAD D, Reg1
ADD Reg1, #1                K.S.I
STORE Reg1, D

P2: ...........
```

Load D, Reg2

ADD Reg2, #2 K.S.2

STORE Reg2, D

.....

Pošto su P1 i P2 **konkurentni** procesi nikakve pretpostavke o brzini izvršenja ovih procesa ne možemo učiniti. Usvajajući da je početna vrednost promenljive D bila 0, nakon izvršenja procesa P1 i P2 može se desiti da D ima vrednost 1, 2 ili 3, u zavisnosti od relativnih vremena izvršenja instrukcija. Da bi izbegli ovu situaciju, tj. da bi obezbedili da vrednost D bude uvek 3 nakon izvršenja procesa P1 i P2, potrebno je da svaki od ovih segmenata programa bude nedeljiva celina i da procesi P1 i P2 pristupaju promenljivoj D uzajamno isključivo. Kodni segment (deo programa) koji se obraća deljivoj promenljivoj predstavlja tvz. **Kritičnu sekciju** (KS)

Uzajamno isključivo pravo pristupa kritičnoj sekciji može se koncepcijski implementirati **mehanizmom zaključavanja**. Naime, možemo deljivoj promenljivoj D pridružiti ključ **K** nad kojim se mogu obavljati dve operacije **LOCK(K)** i **UNLOCK(K)** koje postavljaju K na 1 (zaključavanje) odnosno na 0 (otključavanje). Proces može izvršiti **LOCK(K)** samo aka je K otvoren (tj. postavljen na 0). Rezultat izvršenja LOCK(K) je postsvljanje K na 1, i na taj način se sprečava da drugi proces obavi LOCK(K) operaciju. Proces koji naidje na zaključanu bravu, mora da čeka. Operacijom UNLOCK(K) vrednost K se postavlja na O. Usvajajući da je K inicijalno postavljeno na 0, krticne sekcije u PI i P2 mogu se sada ovako implementirati na bezbolan način.

```
PI:  LOCK(K)
     Kriticna sekcija 1
     UNLOCK(k)

P2:  LOCK(K)
     Kriticna sekcija 2
     UNLOCK(K)
```

U prethodnom primeru kriticne seskcije KS1 i KS2 trebalo je izvršavati uzajamno isključivo ali u bilo kom redosledu.

Uslovne kritične sekcije

Posmatrajmo sada drugu situaciju: da su P1 i P2 **kooperativni**, **komunicirajući**, **iterativni** procesi i da P1 generiše podatke za P2 i pamti ih u D. P2 čita D. Zbog jednostavnosti, usvojimo da D sadži samo jedan podatak. Potrebno je da obezbedimo da P1 čeka pre upisa dok P2 ne pročita prethodni podatak, i da P2 pošto pročita podatak čeka dok P1 ne upiše novi podatak. Tj. potrebno je da P2 ne pročita dva puta isti podatak, ili da P1 ne upiše novi podatak preko podatka koji još nije pročitan. Ovakva sinhronizacija zove se **uslovna sinhronizacija**. I ovaj vid sinhronizacije moguće je koncepcijski ostvariti **mehanizmom zaključavanja**. Sada je potrebno uvesti dva ključa (brave) K1 i K2. P1 otključava K2 da signalizira P2 da je upisao novi podatak. P2 otključava k1 da signalizira P1 da je uzeo prethodno zapamćeni podatak.

Usvajajući da je inicijalno K1=O I K2=1, kodni segmenti u P1 i P2 izgledaju ovako:

PI: LOCK(K1)

generisanje podatka

UNLOCK (K2)

P2: LOCK(K2)

konzumiranje podatka

UNLOCK(K1)

Iz prethodnog je očigledno da **LOCK** operacija sadrži dve aktivnosti: testiranje k i postavlanje vrednosti K na 1. UNLOCK sadrži samo jednu opraciju tj. postsvljanje K na O.

Da bi **LOCK mehanizam** pravilno funkcionisao potrebno je da operacije testiranja i postavljanja budu nedeljive (tj. jedinstvene). Jedan od načina realizacije LOCK mehanizma je pomocu **TEST AND SET** (**TAS**) instrukcije, koja je prvo implementirana kao deo skupa instrukcija sistema IBM 360 (1964), a zatim i kod IBM 370 (1981). Naredborn **TEST AND SET** se izvršavaju sledece operacije kao jedna instrukcija:

X=TAS(K):

X:=K

If X=O then K:=1;

Nedeljivost ove operacije se obično postiže tako što procesor zadržava magistralu dok se ciklus ne zavrsi. Korišćenjem TEST AND SET naredba LOCK(K) se može realizovati na sledeći način:

LOCK(K);

Repeat (x=TAS(k)) until x=O

LOCK instrukcija koja koristi **TAS** ima za posledicu da proces koji pokušava da uđe u kritičnu sekciju bude stalno upošljen testiranjem zajedničke promenljive. Ovo se zove **busy–waiting** (upošljen čekanjem) ill **spin lock**.

Proces koji naidje na zatvorenu bravu ne napušta procesor, već stalno testira vrednost ključa. To ima za posledicu da se procesorski ciklusi beskorisno troše, povećava saobraćaj na magistrali i broj (beskorisnih) obraćanja memoriji. Dakle, rasipanje vrednih resursa zar ne ?

Semafori

Naobrojani nedostaci mogu se rešiti korišćenjem posebne strukture podataka **SEMAFORA**. Semafor je celobrojna promenljiva koja može uzeti samo nenegativne vrednosti. Jedine operacije, izuzev inicijalizacije, koje su dozvoljene nad semaforima su **P** i **V** opracije (neki ih zovu **WAIT** i **SIGNAL**). Ove operacije su nedeljive, tako da se ne može desiti da više procesa izvršava ove operacije nad istim semaforom u isto vreme. Semafore je kao sredstvo sinhronizacije predlozio Danski naučnik **Dijkstra** (1968). Nazivi operacija **P** i **V** poticu od danskih reči **Passern** (proći) i **Vrygeven** (napustiti, osloboditi). Svakom semaforu pridružena je lista procesa koji čekaju na ulazak u kritičnu oblast. Ove

liste se najčešće implementiraju kao redovi čekanja. P i V operacije nad semaforom S imaju sledece dejstvo:

P(S): if S>0 then

S:=S–1

Else (blokirati proces koji je izvrsio P(S) i smestiti ga u red čekanja pridružen semaforu S)

V(S): S:=S + 1

if (red čekanja pridružen semaforu S nile prazan)

then (selektovati jedan proces iz reda čekanja i smestiti ga u red spremnih)

Izvršenje **P(S)** će blokirati proces ako je semafor zatvoren. Blokirani proces neće okupirati procesor kao u slucaju **TEST AND SET**. Procesor može izvršavati neki drugi proces koji je spreman za izvrsenje.

Izvršenje **V(S)** će proces koji je bio blokiran prevesti u stanje spreman. Na ovaj način je eliminisan **busy–waiting** problem.

Semafor može biti **binarni** – uzimati samo vrednosti 0 ill 1, ill **brojni** (generalni) uzimati vrednosti 0 do N. Osnovna razlika izmedju binamog i brojnog semafora je što kod opšteg semafora više procesa može obaviti P operaciju i nastaviti sa izvršenjem.

Prethodni primeri zahtevane sinhronizacije mogli bi se korišćenjem semafora napisati na sledeći način:

1)

S:=1 (inicijalno otvoren semafor)

P1: P(S)

Kritična sekcija 1

V(S)

P2: p(S)

Kritična sekcija 2

V(S)

2) Potrebna su dva semafora SI i S2. Inicijalno je S1:=1, S2:=O

P1: p(S1)

Generisanje podataka

V(S2)

P2: p(S2)

Konzumiranje podataka

V(S1)

Semafor kao sredstvo sinhronizacije ima tu prednost da se lako može realizovati, ali ima i niz nedostataka.

Prvo, semafori obezbeđuju veoma primitivan oblik sinhronizacije procesa i čine paralelni program nejasnim i podložnim greškama. Izostavljanje P ili V operacije ili izvršenje P operacije nad jednim a V nad drugim semaforom, može imati katastrofalne posledice jer uzajamno isključivo pravo pristupa više nije obezbeđeno. Ako procesi u sistemu imaju različite prioritete, sinhronizacija pomoću semafora postaje veoma komplikovana i nerazumljiva. Semaforski programi su nepregledni izmedju ostalog i zato što svaki proces zadržava za sebe tekst svoje kritične oblasti.

MONITORI

Drugi način koji omogućuje mnogo strukturniji prilaz problemu sinhronizacije i komunikacije je korišćenje **monitora**. Za razliku od semaforskih programa kod kojih svi procesi zadržavaju tekst svoje kritične oblasti, sada su sve kritične sekcije, tj. naredbe koje se obraćaju deljivim promenljivim, sadržane u monitoru. Deklaracija monitora sadrŽI brojne procedure koje definišu operacije nad deljivim resursima (promenljivim). Opšti oblik deklaracije monitora je sledeći:

monitor ime;

deldaracija lokalnih promenljivih za monitor

deldaracija procedura lokalnih za monitor

begin

inicijalizacija lokalnih promeljivih

end ime;

Monitorske procedure su u stvari kritične sekcije. Monitorske procedure mogu pristupati samo promenljivim koje su lokalne za dati montor. Ovim lokalnim promenljivim nesme se pristupati van monitora. Ako se ovo poštuje, može se garantovati uzajamna isključivost kod pristupa deljivim promenljivim.

Kada proces želi da pristupi deljivom resursu, kao što je globalna promenljiva ili deljivi hardverski resurs, on to mora učiniti pozivom odgovarajuce monitorske procedure. U jednom trenutku samo jedan proces maže izvršavati neku monitorsku proceduru i ona se mora završiti pre nego što se dozvoli da neki drugi prices pozove neku monitorsku proceduru. Treba napomenuti da sam monitor ne prouzrokuje nikakvu akciju u sistemu. On je jednostavno skup procedura koje se mogu izvrsavati u pojedinim procesima.

Da bi proces pozvao neku monitorsku proceduru potrebno je da navede ime monitora, ime procedure i listu stvamih parametara, tj.

ime_monitora.ime_procedure (stvami parametri)

Nakon deklaracije monitor inicijalizuje svoje lokalne podatke, i svi dalji pozivi koriste vrednosti lokalnih promenljivih dobijenih završetkom prethodnog poziva

Da bi se obezbedila potrebna sinhronizadja procesa unutar monitora se deklariše i poseban tip podatka **CONDITION**. Podaci ovog tipa mogu se deklarisati samo unutar monitora. Ovi podaci se koriste kada proces želi da **zakasni** svoje izvršenje ili **probudi** prethodno blokirani proces.

Npr:

Var uslov: CONDITION

Na ovaj način je u stvari deklarisan red čekanja uslov.

Nad promenljivima tipa CONDITION moguće je izvršavati samo dve opracije:

WAIT(uslov) I SIGNAL(uslov).

WAIT (uslov) operacija blokira proces i dodaje ga redu čekanja pridruženom promenljivoj uslov. Ova operacija automatski oslobadja pristup monitoru, taka da drugi procesi mogu da uđu u monitor.

SIGNAL (uslov) omogućava da se aktivira proces koji je bio blokiran zbog *uslov*. Ako nema ni jednog procesa koji čeka, ova naredba je bez dejstva.

Kao moćno sredstvo komunikacije i sinhronizacije monitor je široko prihvaćen i ugrađen u mnoge više programske jezike, kao npr. Concurrent Pascal, Modula–2, MESA, CSP/K (serija jezika izvedena iz PL/I)

Semafori i monitori kao sredstvo sinhronozacije i komunikacije su upotrbljivi ako se procesi izvršavaju unutar jednog globalnog okruženja (npr. multiprocesorski sistem). Medjutim, aka se procesi odvijaju unutar više lokalnih okruženja, npr. u **MIMD** sistemu sa distribuiranom memorijom, procesi ne mogu pristupati zajednickim promenljivim, pa semafori i monitori nisu upotrebljivi. Procesi u ovakvim sistemima komuniciraju **razmenom poruka**.

Razmena Poruka

Primitivne operacije koje omogućuju razmenu poruka su **SEND/RECEIVE** tipa. Proces šalje poruku izvršavajući operaciju:

SEND izraz **TO** odredišni proces

Proces koji prima poruku izvršava operaciju

RECEIVE promenljiva **FROM** izvorni proces

Tehniku slanja poruka prvi je u jezik koji se zove **CSP (Communicating Sequential Processes)** ugradio Hoare, a zatim Brinch–Hansen u jezik DP (Distributed Processes).

Kod **CSP** proces **P** šalje poruku procesu **Q** izvršavajući sledeću naredbu:

Q!(izraz)

Npr:

Q!(x+y).

Proces Q prima poruku od P izvršavajući naredbu:

P?(promenljiva)

Npr:

P?(z).

Nedostatak ovakvog izražavanja je potreba za obostranim imenovanjem procesa. Naime, izvorni proces mora imenovati odredišni, i obnuto. Da bi se ostvarila komunikacija izmedju dva konkurentna procesa procesi moraju proći kroz dva koraka.

Prvi korak je **sinhronizacija**. Ona se odnosi na činjenicu da oba procesa moraju da stignu u tačku gde se zahteva razmena podataka. Ako neki proces stigne pre, mora da čeka. Kada oba procesa stignu u odgovarajucu tačku, sinhronizacija je postignuta.

Sledeći korak je **komunikacija**. Kao moćno sredstvo komunikacije, tehnika slanja poruka je ugradjena u konkurentni jezik **Ada**. Komunikacija u Adi zove se **randevu**.

Randevu

Osnovna konkurentna jedinica izvršenja u prograrnskorn jeziku Ada zove se **TASK**. Task se sastoji iz specifikacijskog dela i tela.

Specifikacijski deo (ako postoji) sadrži **entry** deklaracije koje predstavljaju sredstvo pomoću kojeg jedan zadatak može komunicirati sa drugirn. Svakom **entry** je pridružen red čekanja. **Entry** je sličan specifIkaciji procedure. Međutim, dok se telo procedure izršava u pozivnom tasku, Entry se izvršava u sopstvenom tasku.

Naredba **ACCEPT** specifIcira akcije koje treba izvrsiti kada se pozove **entry**. Opis u **accept** naredbi mora se slagati sa opisom u entry deklaraciji po broju i tipu promenljivih. U sledećem primeru data je deklaracija jednog zadatka koji sadrži entry deklaracije.

```
task M is (specifikacijski deo)
    entry put (S:in INTEGER) (in ukazuje da je promenljiva S ulazna promenljiva)
end M;

task body M is (telo zadatka)
    –deklaracija lokalnih promenljivih za zadatak
begin
.
.
accept put (S:in INTEGER) do
    PRINT (S) (kod randevua)
end put:
.
.
```

end.

Entry deklaracija može sadržati više promenljivih od kojih neke mogu biti ulazne (in), neke izlazne (out), a neke i ulazne i izlazne (in out). Npr.

entry A (I in INTEGER; x: out REAL; b: in out BOOLEAN)

Poziv entry–a na randevu u nekom zadatku izgleda ovako:

ime_zadatka.ime–entry (stvami parametri)

Npr:

M.put(Q).

Randevu – sinhronizacija i komunikacija izmedju procesa ostvaruje se kada se upare **accept** naredba i poziv **entry**–a u nekom zadatku. Ako neki proces prvi stigne u tačku koja zahteva komunikaciju, mora da čeka. Efekat randevu–a je da prouzrokuje izvršenje naredbi izmedju **do** i **end** u **accept** naredbi pozvanog zadatka, za parametre koji su specificirani u pozivu **entry**–a.

Jos jednom naglasimo da se entry izvršava u pozvanom zadatku a ne u pozivnom (kao sto je slučaj kod procedura). Kada se **randevu** odvija parametri se razmenjuju po istim pravilima kao parametri procedura. Zadatak (task) koji poziva entry drugog zadatka se zaustavlja dok pozvani zadatak izvršava >>KOD<< u **accept** naredbi.

Ako nekoliko zadataka poziva isti **entry** oni formiraju **FIFO** red. Pozvani zadatak (zadatak u kame je entry deklarisan) može koristiti **COUNT** atribut entry–a da vidi koliko zadataka trenutno čeka na **randevu**. Ovaj atribut se dodaje imenu entry–a i kao rezultat vraća broj zadataka koji čeka na randevu sa odgovarajućim entry–jem.

Jedan zadatak može sadržati više entry deklaracija i može komunicirati sa više drugih zadataka. Pošto su zadaci koji se izvrsavaju **asinhroni** ne može se predvideti redosled kojim zadaci komuniciraju. Da bi se uvela nedeterminisanost u program u Adi je na raspolaganju naredba **SELECT.** Ova naredba se sastoji od niza alternativa koje se odnose na prihvatanje (accept) nekog poziva od kojih će samo jedan u datom trenutku biti prihvaćen. Npr:

Select:

accept MJ (fomlalni parametri) do

.....

end M1

or

accept M2 (fomJalni parametri) do

.....

end M2

or

.

.

.

```
end select;
```

Uslovno izvršenje **select** alternative može se postići uvodjenjem when naredbe,

npr:

```
select
            when uslov =>
            accept M1 (formalni parametri) do
            .........
            end M1
or
            when M2 'COUNT >0 =>
            accept M2(formali parametri) do
            .........
            end M2
end select;
```

Select naredba može imati i **else** deo koji se izvršava ako ni jedan od uslova u select naredbi nije ispunjen.

Zadatak može biti deklarisan u okviru tela glavnog program a (kao recimo procedure u Pascalu), ali može biti napisan i kao nezavisna celina koja se nezavisno prevodi, Izgled glavnog programa u Adi je sledeci:

program Ime;

–deklaracija konstanti

–deklaracija tipova

–deklaracija taskova i procedura

–telo taska 1;

–telo taska 2;

.....

–deldaracija promenljivih

begin

.....

end.

Ako se zadaci pišu kao nezavisne celine koje se posebno prevode onda u glavnom programu postoji samo deklarativni deo zadatka koji sadrži **entry** specifIkacije, a umesto tela zadatka pise se:

task body ime is SEPARATE;

Npr.

Program MAIN;

task T1 is

entry Ml (fomalni parametn)

entry M2 (fomalni parametri)

end T1;

task T2 is

entry MMl(fomalni parametri)

end T2;

task body Tl is SEPARATE;

task body T2 is SEPARATE;

begin

.....

end.

Zadaci koji se pišu kao nezavisne celine moraju da sadrže oznaku pripadnosti glavnorn programu.

Npr.

SEPARATE (MAIN)

task body Tl is

–deldaracija lokalnih promenljivih

–deklaracija procedura

.....

begin

.....

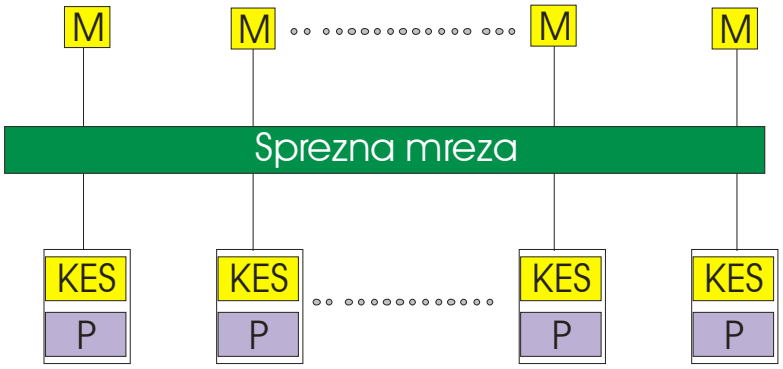
end T1;

KEŠ KOHERENCIJA

Da bi se kod **multiprocesorskih** sistema otklonili problemi kao što su:

- Sudari kod pristupa zajedničkoj memoriji
- Konflikti pri komunikaciji (ako više procesora zahteva isti spržni put)
- Latentnost pristupa kroz sprežnu mrežu (kod multiprocesorskih sistema sa velikim brojem procesora sprežna mreža je veoma kompleksna pa je latentnost kod takvih mreža dugačka)

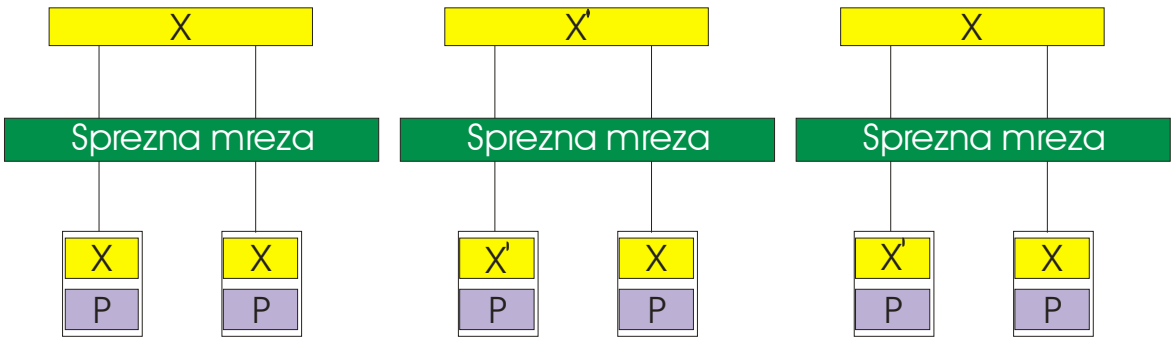
koriste se privatne keš memorije koje se pridružuju svakom procesoru, pa struktura multi–procesorskog sistema ima sledeci izgled:



Može se desiti da u jednom trenutku više keš memorija poseduje kopiju istog podatka iz glavne memorije. Bilo koja lokalna modifikacija kopije podatka u kešu dovešće do **nekonzistentnosti** (neusaglasenosti) memorijskog sistema.

Npr: Ako posmatramo multiprocesorski sistem sa dva procesora, svaki sa privatnim kešom, koji imaju kopiju istog podatka X (slika a), i aka jedan od procesora, recimo P1, modifikuje sadržaj lokacije X na X' u svom kešu, ista vrednost biće automatski upisana u deljivu memoriju aka se koristi >>**write – through**<< (W–T) **politika** za ažuriranje glavne memorije (kod ove politike svaki upis u keš automatski modifikuje i sadržaj glavne memorije). U ovom slučaju **nekonzistentnost** postoji izmedju dve kopije keša (slika b).

Ako se pak koristi >>**write–back**<< (W–B) **politika** kod ažuriranja glavne memorije, onda će i sadržaj glavne memorije biti **nekonzistentan**. Kod W–B politike glavna memorija se ažurira tek kod zamene bloka (slika c).

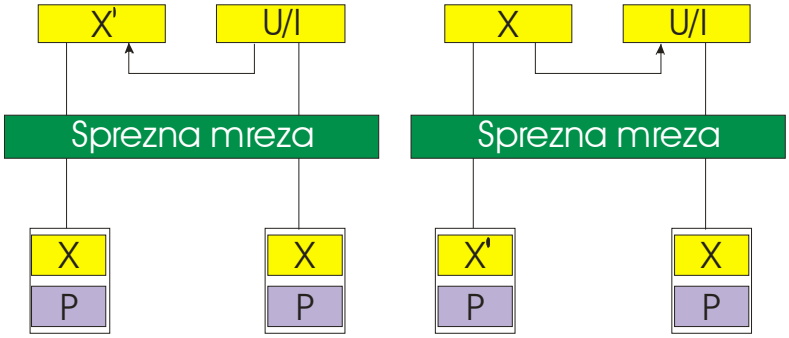


Slika a Slika b (W-T) Slika b (W-B)

Do **nekonzistentnosti** može doći i zbog U/I aktivnosti i migracije procesa u sistemima sa privatnim kes memorijama.

Npr:

Kod većine multiprocesorskih sistema organizovanih oko zajedničke magistrale, U/I procesor je takođe spregnut na magistralu. U slucaju **W–T** politike kod ažuriranja glavne memorije do nekonzistentnoski može daći kada **U/I** procesor puni glavnu memoriju, kao što je prikazano na slici a). U slucaju W–B politike kod ažuriranja glavne memorije do nekonzistentnoski može doći kada se podaci direktno šalju iz memorije u U/I procesor, slika b).



Slika a Slika b

Neki sistemi dozvoljavaju **migraciju procesa**, tj. da procesi budu rasporedjeni različitim procesorima u toku svog života da bi se izbalansiralo opterećenje izmedju procesora. Ako se ova mogućnost koristi u sprezi sa privatnim keš memorijama takodje može daći do nekonzistentnosti podataka.

Npr:

Proces A koji se izvršava na procesoru P1 može promeniti podatke u svom kešu pre nego što bude suspendovan. Ako kasnije proces A migrira na procesor P2 pre nego što glavna memorija bude ažurirana, proces A može uzeti bajatu vrednost iz memorije. Evidentno je da se samo **W–T** politikom kod ažuriranja glavne memorije neće održati konzistentnost memorijskog sistema, jer se ovom tehnikom ne ažuriraju kopije podataka u drugim keš–evima.

Postoji više prilaza za rešavanje ovog problema. Jedan prilaz se zasniva na **hardverski** implementiranim protokolima za postizanje keš koherencije, a drugi na **softverskim** tehnikama.

Softverske tehnike

Nekonzistentnost memorijskog sistema se može sprečiti tako što se **neće keširati** deljivi podaci koji se mogu menjati upisom, a instrukcije i drugi podaci se mogu kopirati u keš. To znači da se podaci na neki način moraju označiti. To može učiniti korisnik korišćenjem viših programskih jezika kao što su Ada, Modula 2, Concurrent Pascal, itd., deklarisati podatke kao deljive (shared) ili nedeljive (local).

Alternativno, multiprocesorski kompajler može automatski klasifikovati podatke kao deljive ili ne.

Nedostatak ove tehnike je netransparentnost multiprocesorske arhitekture za korisnika ili kompajler. Efikasnost ovog pristupa zavisi od mogućnosti jezika da specificira podatke kao deljive ili ne, ili od kompajlera da detektuje takve podatke.

Pošto u praktičnim implementacijama čitava stranica mora biti deklarisana kao >>**cachable**<< ili >>**non–cachable**<<, može doći do interne fragmentacije memorije, ili će se zabraniti keširanje i onih podataka koji se kogu kopirati u keš.

Drugi problem se javlja zbog U/I aktivnosti i migracije procesa. U slučaju U/I aktivnosti svaki keš mora biti poništen >>cache flushing<< pre nego što se dozvoli U/I aktivnost, ili se svi podaci koji su predmet U/I aktivnosti moraju označiti kao >>noncachable<<. Slično, ako je u pitanju migracija procesa, keš mora biti poništen pre migracije ill se mora zabraniti migracija procesa po cenu smanjenja performansi.

Hardverski protokoli keš koherencije

U ovom slučaju se hardverskim mehanizmima detektuju uslovi koji mogu dovesti do **nekonzistentnosti** keš memorija i shodno tome obavljaju akcije koje održavju koherentnost memorijskog sistema. U ovu grupu protokola sadaju:

- Snoopy keš protokoli (njuskala)
- Direktorijumske šeme

Mnogi današnji komercijalno raspoloživi multiprocesori koriste memorijske sisteme organizovane oko zajedničke magistrale. Magistrala je pogodna za održavanje keš koherencije jer kao jedinstveni sprežni put, dozvoljava da svi procesori u sistemu nadgledaju memorijske transakcije. Ako transakcija na magistrali ugrožava konzistentnost podataka u lokalnom kešu, specijalni hardver (keš–kontroler) može preduzeti akcije da poništi lokalnu kopiju (tj. proglasi je nevažećom). Protokoli koji koriste ovaj mehanizam za postizanje koherentnosti zovu se ***snoopy* (njuskala)** protokoli jer svaki keš kontroler "njuska" transakcije drugih keševa.

Druge sprežne mreže, kao npr. Višestepene, nemaju pogodne **snoopy mehanizme** i mehanizme za emisiju svima (broadcast). Kod takvih sistema problem keš koherencije se rešava uz pomoć **direktorijumskih šema**.

Snoopy keš protokoli

Kada su u pitanju ovi protokoli praktikuju se dva prilaza (politike) za postizanje keš koherencije:

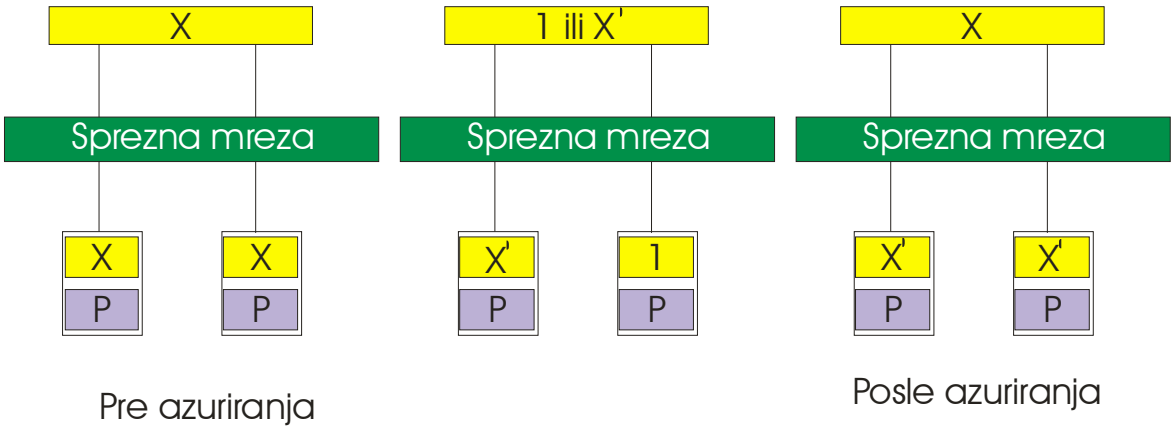
- Invalidacija (poništavanje) pri upisu (**write–invalidate**)
- Ažuriranje pri upisu (**write–update**)

Kod politike **invalidacije pri upisu** konzistentnost većeg broja kopija održava se na sledeći način:

Kada procesor ažurira svoj lokalni blok podataka sve ostale kopije tog bloka u sistemu se poništavaju. Sva naredna ažuriranja od strane procesora koji je izvšio invalidaciju se izvode lokaalno (bez izdavanja komande invalidacije) jer u sistemu postoji samo jedna važeća kopija. Na slici 1 je ovo ilustrovano za slučaj dvoprocesorskog sistema.

Ako procesor P2 izda zahtev za čitanjem podatka koji se nalazi u bloku X', tada keš pridružen procesoru P1 direktno obezbeđuje taj podatak ili se prvo ažurira glavna memorija pa se iz nje pribavlja podatak. Sve u svemu u kešu procesorskog elementa P1 se nalazi jedina kopija podatka X'.

Kod druge politike (**ažuriranje pri upisu**), umesto da se kopije invalidiraju one se **ažuriraju**. Kada će se glavna memorija ažurirati zavisi da li se koristi **write–through** (W–T) ill **write–back** (W–B) politika ažurirania glavne memoriie. Na slici 2 je ova ilustrovano za slučaj dvoprocesorskog sistema.



Snoopy protokol sa invalidacijom pri upisu

Prvi protokol ovog tipa predloženje 1983 god. i zovao se >>**Write-Once<<**. Ovaj protokol koristi **W-T** politiku kod ažuriranja glavne memorije pri **PRVOJ** modifikaciji keš bloka. Nakon prvog upisa memorija se ažurira korišćenjem **W-B** politike. Ovaj protokol razlikje četiri stanja kopije keš bloka:

- **Valid**: (važeća): U kešu je kopija koja je konzistentna (jednaka) sa memorijskom kopijom i nije modifikovana (tj. konzistentna je sa ostalim kopijama, aka postoje, u drugim keševima).
- **Invalid**: Nevažeća kopija (tj. kopija je poništena)
- **Reserved**: (rezervisano) Podatak je u kešu modifikovan samo jednom i keš kopija je konzistentna sa kopijom u glavnoj memoriji, jer je memorija ažurirana korišćenjem **W-T** politike. Ona je jedina druga(različita) sve ostale kopije, aka su postojale su invalidirane.
- **Dirty**(prljava): Keš je modifikovan više puta i kopija je jedinstvena u sistemu. U sistemu može postojati samo jedna prljava kopija.

Da bi se održala konzistentnost memorijskog sistema protokol koristi dva skupa komandi:

- Komande koje izdaje procesor lokalnom kešu (lokalne procesorske komande **P_Read** (komanda za čitanje lokalnog keša), i **P_Write** (komanda za upis u lokalni keš).
- Komande konzistencije koje se izdaju preko zajedničke magistrate: **Read_Blk** (čitanje bloka iz memorije), **Write_Blk** (upis bloka u memoriju), **Write_Inv** (modifikuje lokalnu kopiju i invalidira sve ostale kopije tog bloka koje se nalaze u ostalim keševima), **Read_Inv** (čita blok iz memorije, modifikuje ga i invalidira sve ostale kopije).

Kod obraćanja procesora lokalnom kešu mogu nastupiti sledeći dogadjaji:

- Pogodak kod čitanja
- Promašaj kod čitanja
- Pogodak pri upisu
- Promašaj pri upisu
- zamena bloka

U zavisnosti od toga koji je dogadjaj nastupio kod obraćanja procesora lokalnom kešu, preduzimaju se sledeće akcije:

Pogodak kod čitanja: Obavlja se lokalno i ne uslovljava promenu stanja keš kopije i ne zahteva izdavanje komandi konzistencije.

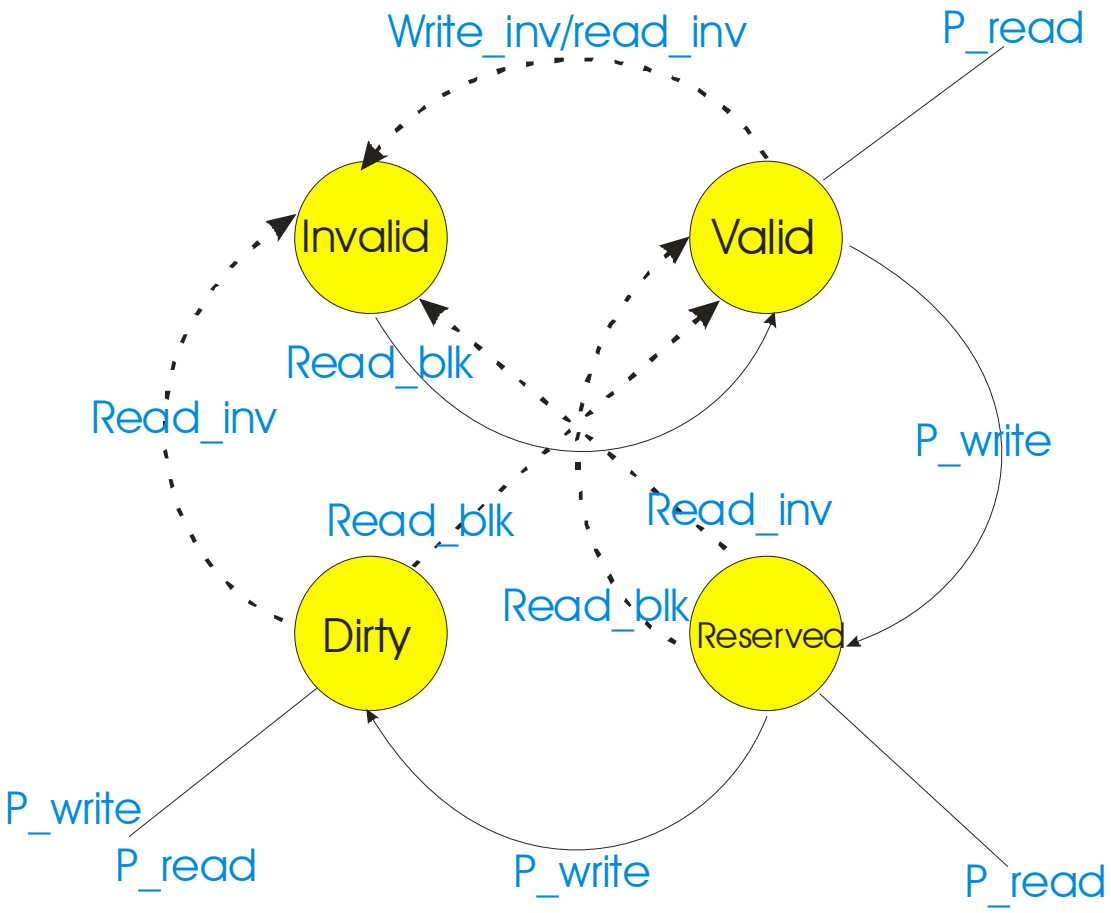
Promašaj kod čitanja: Aka nastupi promašaj kod čitanja i ni jedan keš u sistemu ne sadrži **drty** kopiju, inicira se **Read_Blk** operacija i podatak (tj. ceo blok) se prenosi iz glavne memorije u lokalni keš. Aka postoji prljava kopija, tada će odgovarajući keš kontroler zabraniti čitanje glavne memorije i poslati kopiju kešu koji je izdao zahtev. Obe kopije postaju validne i ažurira se i glavna memorija.

Pogodak pri upisu: Aka nastupi pogodak pri upisu i kopija je u stanju *reserved* ili *dirty* novo stanje kopije je *dirty*. *A*ko je kopija bila u stanju *valid,* tada se svim keševima šalje komanda Write_Inv pomoću koje se invalidiraju sve kopije. Memorijska kopija se ažurira, novo stanje lokalne keš kopije je *reserved*.

Promasaj pri upisu: Kada nastupi promašaj pri upisu procesor mora prvo pribaviti kopiju iz glavne memorije ili iz keša koji ima dirty kopiju. Ova se postize slanjem Read_Inv komande koja će pribaviti i ažurirati blok i invalidirati sve ostale keš kopije. Lokalna kopija nakon ovoga se nalazi u stanju *dirty*.

Zamena bloka: Ako je stanje bloka **dirty**, kod zamene bloka se vrši upis u glavnu memoriju (Write_Blk), a aka je blok invalid, reserved ili valid nema upisa u memoriju kod zamene bloka.

Promene stanja kopija koje se dešavaju u toku primene ovog protokola mogu se predstaviti grafom kao na slici. Punim linijama su oznaceni prelazi koji nastupaju izdavanjem lokalnih procesorskih komandi, a isprekidanim prelazi zbog komandi konzistentcije.



Snoopy protokol sa ažuriranjem pri upisu

Kao primer protokola koji konzistentnost memorijskog sistema odrzava primenom politike ažuriranja pri upisu navešćemo protokol **Firefly** (svitac). Kod ovog protokola postoje tri moguća stanja keš kopije:

- **Valid–exclusive** – postoji sarno jedna keš kopija i onaje konzistentna sa glavnom memorijom.
- **Shared** – postoji više kopija i sve su konzistentne
- **Dirty** – postoji samo jedna kopija i ona nije konzistentna sa glavnom memorijom.

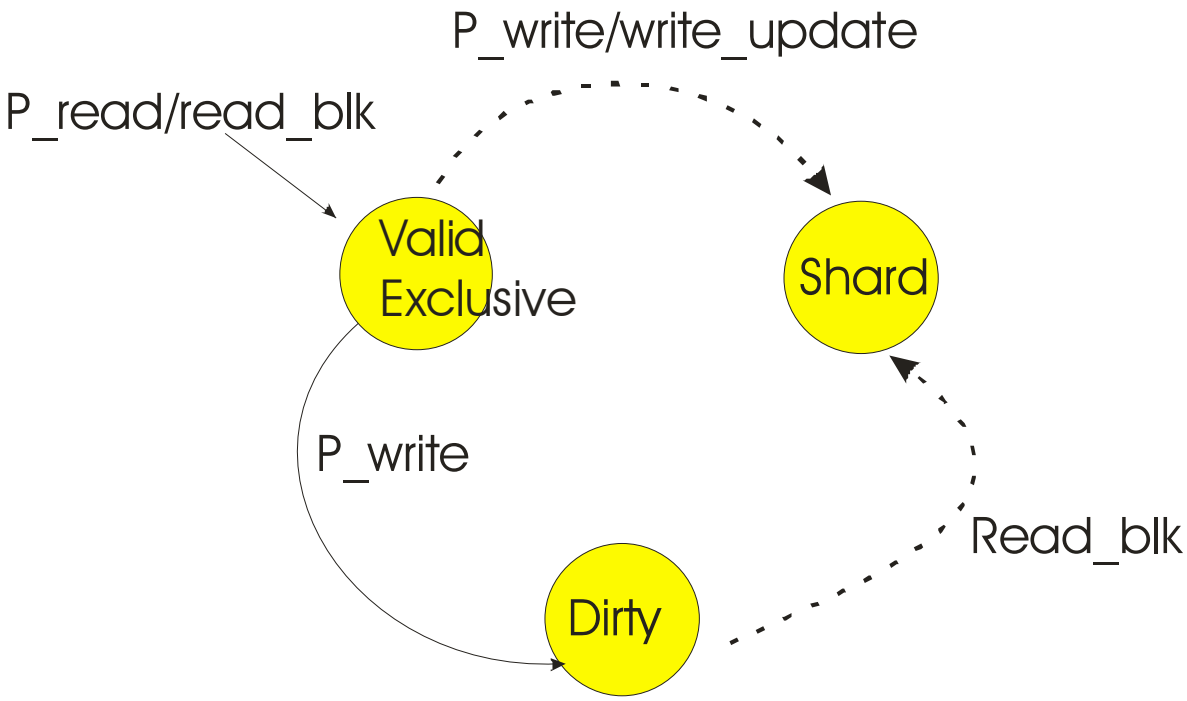
Firefly protokol koristi write–back (**W–B**) politiku za ažuriranje privatnih blokova (to su blokovi koji su u stanju valid–exclusive i Dirty), a **W–T** za deljive (shared) blokove.

Da li je blok deljiv ili privatni, određuje se u toku izvršenja programa. Da bi se održala konzistentnost kod modifikacije deljivog (shared) bloka koristi se **Write–Update** komanda koja ažurira sve kopije u sistemu.

U zavisnosti koji je dogadjaj nastupio kod obraćanja kešu preduzimaju se sledeće akcije:

- **Pogodak kod čitanja**: Obavlja se lokalno i ne uslovljava promenu stanja keš kopije i ne zahteva izdavanje komandi konzistencije.
- **Poromašaj kod čitanja**: Ako postoji **shared** ili **Valid–Exclusive** tada se blok pribavlja iz glavne memorije. Ako postoji dirty kopija tada keš sa dirty kopijom predaje blok podataka, ažurira glavnu memoriju, a rezultujuće stanje kopije je Shared. Ako ni jedan keš ne poseduje kopiju bloka, tada se on pribavlja iz glavne memorije a stanje kopije je Valid–exclusive.
- **Pogodak pri upisu**: Aka je blok **Dirty** ili **Valid–exclusive** tada se upis vrši lokalno i novo stanje je Dirty. Ako je kopija **Shared** sve ostale kopije bloka (uključujući i memorijsku) se ažuriraju.
- **Promasaj pri upisu**: Kopija dolazi iz drugih keševa ili iz glavne memorije ako dolazi iz memorije stanje kopije nakon upisa je dirty. U ostalim slučajevima sve kopije se ažuriraju a novo stanje je shared.
- **Zamena bloka**: Aka je stanje bloka dirty, kod zamene bloka se vrši upis u glavnu mernoriju (Write_Blk). U ostalim slučajevima akcije se ne preduzimaju.

Promene stanja keš kopija koje se dešavaju u toku primene ovog protokola mogu se predstaviti grafom kao na slici. **Punim** linijarna su označeni prelazi koji nastupaju izdavanjern lokalnih procesorskih komandi, a **isprekidanim** prelazi zbog kornandi konzistentcije.


P_write/write_update
P_read/read_blk
Valid
Exclusive
Shard
P_write
Read_blk
Dirty

Direktorijumske šeme

Kada se koriste vešestepene sprežne mreze za gradnju multiprocesorskih sistema sa velikim brojem procesora, **snoopy** protokoli se moraju modifikovati da bi se prilagodili mogućnostima mreže. Naime, pošto je >>**broadcast**<< (emisiju) veoma skupo obaviti kod višestepenih mreža, komande konzistentnosti se šalju samo onim keševima koji imaju kopiju bloka. Da bi se to učinilo potrebne su tačne informacije o tome koje keš memorije imaju kopiju odredjenog bloka. Ove informacije se smeštaju u **direktorijumima** (adresarima). Protokoli koji koriste informacije zapamćene u direktorijumima za održavanje koherentnosti memorijskog sistema zovu se **direktorijumske šeme**.

Svaki memorijski modul sadrži poseban direktorijum koji beleži stanje i prisustvo memorijskog bloka u određenom kešu. Svaki direktorijumski ulaz (linija) sadrži veći broj pokazivača koji ukazuju na keš koji sadrži kopiju bloka i jedan "dirty" bit kojim se ukazuje kada određeni keš ima, ili ne, dozvolu da modifikuje podatak.

Postoje tri tipa direktorijumskih protokola i to:

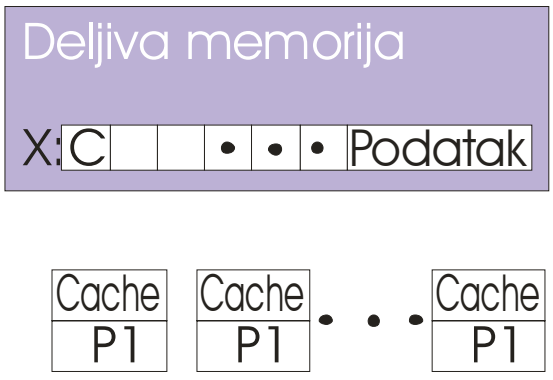
- Protokoli sa potpuno preslikanim adresarima (**Full–map directories**)
- Protokoli sa ograničenim adresarima (**limited directories**)
- Protokoli sa ulančanim adresarima (**chained directories**)

Protokoli sa potpuno preslikanim adresarima

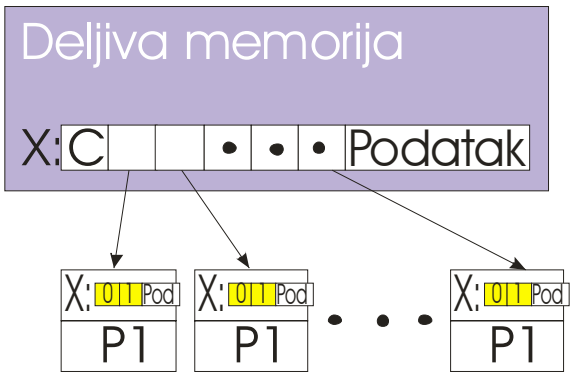
Kod protokola sa potpuno preslikanim adresarima, svaki direktorijumski ulaz sadrži po jedan bit za svaki procesor u sistemu i jedan **dirty** bit. Svaki bit pridružen procesor ukazuje na status bloka u odgovarajućem procesorskom kešu (prisutan ili ne). Ako je **dirty** bit postavljen, tada je jedan i samo jedan procesorski bit postavljen i taj procesor ima pravo da vrši upis u kešovani blok (tj. da ga modifikuje).

Svaki keš direktorijum sadrži dva bita stanja za svaki kešovani blok. Jedan bit ukazuje da li je blok **važeći**, a drugi da li se u važeći blok može vršiti upis. Keš koherentni protokol mora držati bitove stanja u memorijskom direktorijumu i u kešu konzistentnim.

Promene stanja u adresaru u zavisnosti od akcija koje se preduzimaju mogu biti prikazane sledecom slikom:



Slika :a



Slika :b

Na slici a) ni jedan keš ne sadrži kopiju lokacije X. Na slici b) procesori P1, P2 i Pn su zahtevali kopiju podatka X. Tri bita u odgovarajućem adresarskom ulazu u glavnoj memoriji koji ukazuju na prisustvo bloka u određenom kešu se postavljaju. U keš adresaru se postavljaju odgovarajući bitovi stanja da ukažu na važeću kopiju. U adresaru u glavnoj memoriji "dirty" bit je obrisan (c), i ukazuje da ni jedan procesor nema pravo da modifikuje kopiju podataka u svom kešu.

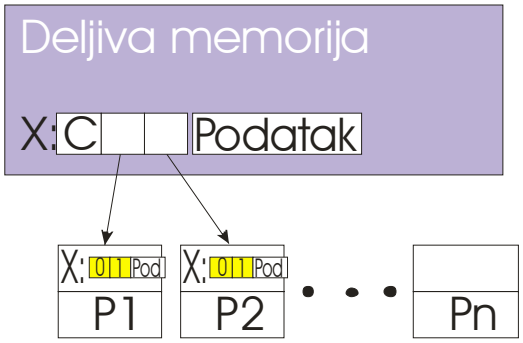
Analizirajmo sada šta se dešava aka jedan procesor, recimo Pn, izda komandu za upis u svoj keš (Cn):

- Keš Cn detektuje da je blok koji sarži lokaciju X važeći, ali da procesor nema pravo upisa.
- Keš kontroler Cn upućuje zahtev za modifikacijom odgovarajucem memorijskom modulu koji sadrži lkaciju X i zaustavlja svoj procesor, Pn.
- Memorijski modul izdaje zahtev za invalidacijom keševa C1 i C2
- Keševi C1 i C2 primaju invalidacione zahteve, postavljaju odgovarajuće bitove koji ukazuju da je blok koji sadrži lokaciu X nevazeći i vraćaju potvrdu o priznavanju zahteva nazad memorijskom modulu.
- Memorijski modul prima potvrdu o invalidaciji, postavlja "dirty" bit, briše pokazivače na keševe C1 i C2 i predaje dozvolu za upis kešu Cn.
- Keš Cn prima poruku o dozvoli upisa, ažurira stanje u kešu i aktivira procesor Pn.

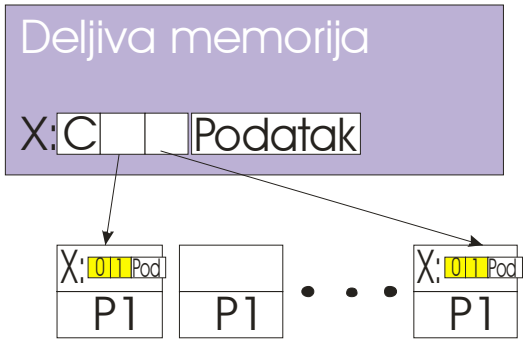
Ovakvim redosledom akcija protokol garantuje da će memorijski sistem ostati konzis tentan. Nedostatak ove adresarske šeme je što se za svaki memorijski blok u adresaru zahteva po jedan bit za svaki procesor. Ako ima N procesora svaki direktorijumski ulaz ima N pokazivača. Sa porastom broja procesora u sistemu, proporcionalno se povecava i adresarski prostor (tj gubitak memorije zbog vođenja evidencije u adresarima). Isto vazi i aka se povećava veličina memerije. Ovi problemi se mogu rešiti aka se koriste ograničeni adrsari.

Protokoli sa ogranicenim adresarima

Kod šema sa ograničenim adresarima ograničava se broj jednovremenih kopija jednog memorijskog bloka (tj. samo određeni broj procesora maže sadržati kopiju bloka u svom kešu). Ovaj protokol je sličan protokolu sa potpuno preslikanim adresarima, izuzev kada više keševa nego što je dozvoljeno zahteva kopiju jednog bloka podataka. Svaki direktorijumski ulaz u glavnoj memoriji sadrži određeni (ograniceni) broj pointera na procesore koji sadrže kopiju bloka i jedan "dirty" bit koji ukazuje da li se blok može modifikovati. Na primer, aka je broj kopija ograničen na dve, i aka se kopije nalaze u keševima procesora P1 i P2, tada će direktorijumski ulazi u adresaru glavne memorije sadržti pointere na procesore P1 i P2, kao što je prikazano na Slici a. Ako sada procesor Pn zahteva kopiju istog bloka podataka, kontroler glavne memorije mora invalidirati kopiju bloka u kešu procesora P1 ili P2 i zameniti pokazivač, kao što je prikazano na Slici b.



Slika :a

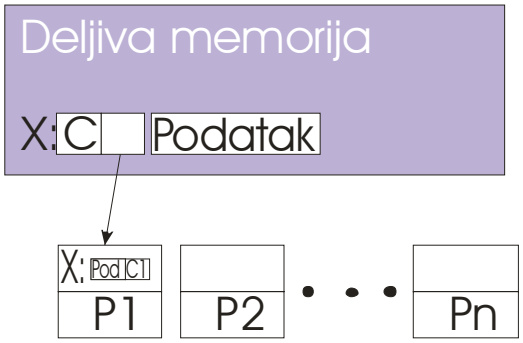


Slika :b

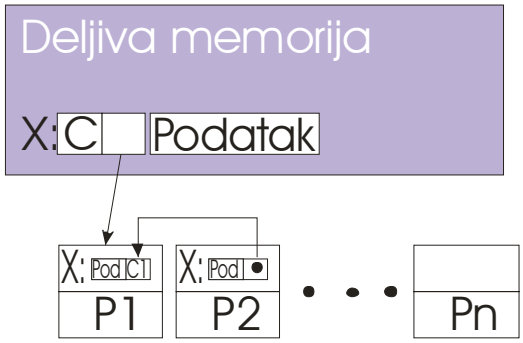
Protokoli sa ulančanim adresarima

Ovi protokoli dozvoljavaju proširljivost sistema bez ograničavanja broja kopija jednog memorijskog bloka bloka. Ovaj tip adresarske šeme zove se ulančani adresari. Jer čuva trag o kopijama bloka preko lanca pokaĐivača. Formiranje adresarskog lanca objašnjeno je na sledećoj slici.

Pretpostavimo da u sistemu ne postoji kopija bloka sa lokacijom X. Aka P1 čita lokaciju X, glavna memorija šalje kopiju bloka kešu procesora P1 i ujedno pokazivač na kraj lanca CT (Chain Termination). A u adrsaru glavne memorije čuva se pokazivač na procesor P1 slika a). Ako sada P2 zahteva čitanje lokacije X, glavna memorija šalje kopiju bloka kešu procesora P2 i pokazivac na P1. Sada se u adrsaru glavne memorije cuva pokazivac na procesor P2 slika b). Ponavljanjem ovih koraka svi keševi mogu dobiti kopiju bloka.



Slika :a



Slika :b

Pretpostavimo sada da jedan od procesora, recimo P3, zahteva da modifikuje sadržaj lokacije X. Da bi se održala konzistentnost memorijskog sistema potrebno je da se komanda invalidacije prosledi kroz lanac. Kontroler glavne memorije ne daje dozvolu modifikacije procesoru P3 sve dok ne primi potvrdu o invalidaciji od procesora koji sadrži CT pointer.

S obzirom da se informacija o invalidaciji prenosi od jednog do drugog keša u lancu, ovaj protokol se zove i "gossip" (tracarenje) protokol.

VEKTORIZACIJA (ZADACI)

Vektorizacija je proces konverzije sklarnih operacija u petlje u ekvivalentne vektorske naredbe. Paralelizacija ima za cilj konverziju sekvencijalnog koda u paralelni oblik tj. omogućava paralelno izvršavanje programa uz pomoć više procesora. Vektorizacija se vrši optimizirajućim kompajlerom automatski ili poluautomatski)uz direktive programa) i taj kompajler se naziva vektorizujući kompajler. Slično njemu, paralelizujući kompajler generiše paralelni kod tj. pretvara sekvencijalni u paralelni kod automatski ili poluautomatski.

Kod vektorskih računarskih sistema protočbnost je veoma duboka, pa se dešava da se prethodna iteracija ne izvrši a da sledeća počne sa izvršavanjem. Može se desiti da se rezultat predhodne iteracije ne upiše, a da se počne izvršavanje sledeće iteracije. Ovakve zavisnosti se nazivaju Loop Carry zavisnosti. Ovakav hazard vektorski procesor ne detektuje pa ove zavisnosti mogu sprečiti vektorizaciju petlje. Ako postoji RAW zavisnost unutar same itracije tu zavisnost detektuje običan hardver koji sprečava izdavanje instrukcije ako ona izaziva RAW hazard sa perthodnom instrukcijom. WAW i WAR zavisnosti na nivou petlje nisu bitne jer se otklanjaju jednostavnim preimenovanjem.

Vektorizacija ugnježdenih petlji

Programi koji sadrže ugnježdene petlje imaju onoliko iterativnih indeksa koliko petlji. Zavisnosti vektora u indeksnom prostoru utiču na mogućnost odnosno nemogućnost vektorizacije petlji.

Uvodi se pojam vektora zavisnosti **d** koji je definisan kao:

$$d = f(I)^T - g(I)^T$$

gde je f(I) uređena n–torka indeksa generisane promenljive a g(I) uređena n–torka indeksa korišćene promenljive.

Na osnovu svih indeksa zavisnosti formira se matrica zavisnosti:

$$D = [d_1 d_2 d_3 ..... d_n]$$

Ove matrice se koriste u transformacijama kod paralelizacije i vektorizacije petlje.

Na osnovu vektora zavisnosti definiše se i pravac zavisnosti:

$$\Pr avaz\_zavisnosti = \begin{Bmatrix} < & d_i > 0 \\ = & d_i = 0 \\ > & d_i < 0 \end{Bmatrix}$$

Ako vektorizacija nije moguća u unutrašnjoj petlji može se pokušati sa zamenom petlji ili nekom drugom transformacijom nad indeksnim skupom.

Postoje tri vrste elementarnih transformacija. Ove tri transformacije se opisuju na osnovu matrice transformacija T.

Prva transformacija:

$I_1 = e_1, n_1$

$I_2 = e_2, n_2$

.

.

.

$I_m = e_m, n_m$

Ako hoćemo da zamenimo mesta petljama I_i i I_i ta transformacija se opisuje pomoću matrice transformacije dimenzija mxm koja se dobija kada se jediničnoj matrici I_{mxm} zamene mesta J–toj i K–toj vrsti.

NPR:

Za m=2

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Na primer:

```
Do   I,1,N
Do   J=1,N
     A(J)=A(J)+C(I,J)
```

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} J \\ I \end{bmatrix}$$

U=J U=1,n

V=i V=1,n

```
Do   U,1,N
Do   V=1,N
     A(U)=A(U)+C(V,U)
```

Očito je da se zamenom nije ništa pokvarilo u petlji.

Druga transformacija:

Ako po nekom iterativnom indeksu hoćemo da primenimo ovu transformaciju ona se dobija kada se u jediničnoj matrici u I–toj vrsti dijagonalni element zameni sa –1.

```
Do I=1,N
DoJ=1,N
```

A(I,J)=A(I−1,J+1)

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} I \\ -J \end{bmatrix}$$

U=I U=1,n

V=−J V=−n,−1

Do U=1,N

Do V=−N,−1

A(U,−V)=A(U−1,−V+1)

Teća transformacija:

Treća transformacija se naziva krivljenje. Naime, tada se vrši krivljenje jednog iterativnog indeksa u odnosu na drugi za faktor f. Ako imamo (P_1,.....P_{i-1},P_i,....,P_{j-1},P_J,P_{J+1},.....,P_n) i ako primenimo krivljenje petlje I_J u odnosu na I_i za vektor f dobija se:

(P_1,.....,P_i,....,P_{j-1},P_J+fP_i,P_{J+1},.....,P_n)

Transformacija deluje jako složeno, ali se može vrlo lako predstaviti pomoću matrice transformacije:

$$\begin{bmatrix} U \\ V \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} I \\ J \end{bmatrix} = \begin{bmatrix} I \\ J+2I \end{bmatrix}$$

U=I U=1,N I=U

V=J+2I V=2U+1, 2U+N J=V−2U

Od ugnježdenih petlji:

Do I=1,N

Do J=1,N

A(I,J)=A(I,J−1)+A(J−1,J)

Dobiće se:

Do U=1,N

Do V=2N+1,2N+N

A(U,V−2N)=A(U,V−2N)+A(U−1,V−2U)

Da bi ove transformacije bile validne, mora da važi:

$T \cdot d \geq 0$ za svaki vektor d.

Ovaj uslov je bitan da bi se sačuvale zavisnosti koje postoje u redosledu izvršavanja instrukcija.

Na primer:

```
Do      I=1,4
Do      J=1,3
        A(I,J)=A(I,J)+A(I-1,J+2)
```

$d_1 = [0 \quad 0]^T$

Prva nula u ovom vektoru dobija se tako što se oduzmu vrednoti označene plavom bojom, a druga vrednosti označene crvenom bojom.

$d_2 = [1 \quad -2]^T$

Prva nula u ovom vektoru dobija se tako što se oduzmu vrednoti označene plavom i zelenom bojom, a druga vrednosti označene crvenom i ljubičastom bojom.

Dakle, ovako smo dobili matricu D:

$$D = \begin{bmatrix} 0 & 1 \\ 0 & -2 \end{bmatrix}$$

ako po nekom iterativnom indeksu postoje zavisnosti > ili < onda je vektorizacija nemoguća.

Kada bi se ove petlje izvršavale bez ikakve transformacije, dobilo bi se:

Za I=1 i J=1: $A_{11}=A_{11}+A_{03}$

Za I=1 i J=2: $A_{12}=A_{12}+A_{04}$

Za I=1 i J=3: $A_{13}=A_{13}+A_{05}$

Za I=2 i J=1: $A_{21}=A_{21}+A_{13}$

Za I=2 i J=2: $A_{22}=A_{22}+A_{14}$

Za I=2 i J=3: $A_{23}=A_{23}+A_{15}$

I T D

Semafori

Sinhronizacija paralelnih procesa je jedna od osnovnih kategorija u paralelnom programiranju. Njom se rešavaju i definišu vremenske zavisnosti između procesa, odnosno može se reći prenose se vremenski parametri koji kontrolišu izvršavanje procesa.

Semafori su jedna tehnika kojom se rešava problem sinhronizacije paralelnih procesa. Semafor je definisan sa 2 nedeljive operacije P i V, i pridružuje mu se operacija inicijalizacije semafora. Najčešće se realizuje pomoću jedne zajedničke celobrojne promenljive, koja može uzimati samo nennegativne vrednosti.

Kada se semafor realizovan preko deljive promenljive, P i V se mogu definisati na sledeći način:

```
P(S):

if S>0 then

    S:=S–1;

Else

    (Suspendovati tekući proces i staviti ga u red za čekanje);


V(Ss):

if (POSTOJI PROCES U REDU ZA čEKANJE) then

    (probuditi proces);

Else

    S :=S+1;
```

Promenljiva S je deljiva što zanači da se iz svih procesa može pristupiti toj promenljivoj, ali samo preko operacija P(S) i V(s).

Ove dve operacije su nedeljive, što znači da proces ne može piti prekinut u toku jedne od njih, a takođe nad promenljivom S se u jednom trenutku može izvršiti samo jedna operacija (P(S) ili V(S)) bez obzira u kom i u koliko procesa se javljaju.

Semafor može biti binarni (S=0 ili 1) ili brojni (S= 0 ..... M)

Monitori

Videli smo da se sinhronizacija i komunikacija procesa u sistemima sa deljivom memorijom može obaviti korišćenjem semafora.

Semafori su rešavali probleme pristupa zajedničkim resursima (deljive promenljive u zajedničkoj memoriji ili deljivi hardverski resurski) od strane onih procesa koji se paralelno izvršavaju.

Međutim semafori imaju jednu veliku manu i pored toga što je na prvi pogled njihovo korišćenje efikasno. Ta mana se sastoji u činjenici da su naredbe za pristup semaforima rasturene po celom kodu. U slučaju greške u sinhronizaciji trebalo bi pregledati ceo kod svih procesa i analizirati, vrlo komplikovane vez između njih. Takođe, dodavanje naprimer još jednog procesa koji pristupa istom resursu može zahtevati promenu koda u svim procesima, što je naravno suludo.

Problem sinhronizacije i komunikacije paralelnih procesa se pomoću monitora rešava mnogo elegantnije na struktuiran način. Ovde je primenjen centralistički pristup, naime, monitor nije rasut po celom kodu i mnogo je lakše održavati takav kod.

Zadatak br. 1

Kakve zavisnosti postoje između narebi u sledećoj petlji. Da li je petlju moguće vektorizovati i ako nije moguće, planiranjem redosleda naredbi učiniti to mogućim.

```
FOR (i=1, i<=100; i++)
(
A(I)=A(I)+B(I);
B(I+1)=C(I)*D(I);
)
```

Rešenje zadatka br. 1

Za I=1

A(1)=A(1)+B(1)

B(2)=C(1)*D(1)

Za I=2

A(2)=A(2)+B(2)

B(3)=C(2)*D(2)

Primećujemo da ovde postoji Loop Carry zavisnost i to između B(2) i B(2).

Dakle, sada bi trebalo izmeniti program tako da se otkloni ova zavisnost:

```
A(1)=A(1)+B(1);
For *I=1; I<=99, I++)
(
B(i+1) = C(I)*D(i);
A(I+1)=A(I+1)+B(I+1);
)
B(101)=C(100)*D(100);
```

Dakle, sada je petlja skraćena na: 1–99 a ispred i iza petlje su izbačene dve naredbe. Ako malo bolje razmislimo, videćemo da je petlja ostala ista, a da sada nema Loop Carry zavisnosti. To možemo i da proverimo:

Za I=1

B(2)=C(1)+D(1)

A(2)=A(2)+B(2)

Za I=2

B(3)=C(2)+D(2)

A(3)=A(3)+B(3)

Dakle, otklonjena je RAW zavisnost između iteracija a postoji u okviru jedne iteracije što se hardverski otklanja, tako da smo mi naš posao odradili.

Zadatak br. 2

Otkloniti Loop Carry zavisnosti ako postoje i u tom slučaju vektorizovati sledeću petlju.

```
FOR (I=1; I<100; I++)
(
A(I)=B(I)+C(I);
B(I)=A(I)+D(I);
A(I+1) = D(I)+E(I);
)
```

Rešenje zadatka br. 2

Za I=1 imamo:

```
A(1)=B(1)+C(1);
B(1)=A(1)+D(1);
A(2) = D(1)+E(1);
```

Za I=2 imamo:

```
A(2)=B(2)+C(2);
B(2)=A(2)+D(2);
A(3) = D(2)+E(2);
```

Uočavamo da postoji zavisnost između A(2) i A(2). Sada treba pokušati promeniti raspored linija tako da se ova zavisnost izgubi tj. da ne postoji nikakva zavisnost.

```
A(1)=B(1)+C(1)
B(1)=A(1)+D(1)

FOR (I=1; I<99; I++)
(
A(I+1)=D(I)+E(I)
A(I+1)=B(I+1)+C(I+1);
B(I+1)=A(I+1)+D(I+1);
)
A(100)=D(99)+E(99)
```

Dakle, sada neće biti Loop Carry zavisnosti.

Zadatak br. 3

Otkloniti Loop carry zavisnosti u sledećoj petilji

```
FOR (I=6; I<=100; I++)
(
Y(I)=Y(I−5) + Y(I)
)
```

Rešenje zadatka br. 3

Za I=6 Y(6)=Y(1)+Y(6)

Za I=7 Y(7)=Y(2)+Y(7)

.

.

.

Za I=11 Y(11)=Y(6)+Y(11)

Za I=12 Y(12)=Y(7)+Y(12)

Crvenom i plavom bojom sozačiću loop carry zavisnosti. Primetićete da se zavisnosti javljaju na međusobnom rastojanju od I=5. Dakle, Linija I=6 vezana je sa linijom I=11. Dakle, 11−6=5. Ovo nas usmerava na ideju da, ukoliko želimo da uklonimo zavisnosti između linija, petlju treba prepraviti tako da se izvršava 5 puta (jer bi šesto izvršavanje dovelo do loop carry zavisnosti).

Dajle, petlja se treba izvršiti za I=1 do 100. pošto se ona može izvršiti maksimalno 5 puta, zaključujemo da telo petlje treba da bude dugačko čitavih 20 naredbi (20*5=100).

```
FOR (I=6; I<=10; I++)
(
Y(I)=Y(I−5) + Y(I);
Y(I+5)=Y(I) + Y(I+5);
Y(I+10)=Y(I+5) + Y(I+10);
Y(I+15)=Y(I+10) + Y(I+15);
Y(I+20)=Y(I+15) + Y(I+20);
Y(I+25)=Y(I+20) + Y(I+25);
Y(I+30)=Y(I+25) + Y(I+30);
.
.
.
Y(I+90)=Y(I+85) + Y(I+90);
```

)

Ovako organizovanom petljom će se za I=6 izračunati sledeće vrednosti:

Y(6)

Y(11)

Y(16)

Y(21)

.

.

.

Y(96)

Ovako organizovanom petljom će se za I=7 izračunati sledeće vrednosti:

Y(7)

Y(12)

Y(17)

Y(22)

.

.

.

Y(97)

Ovako organizovanom petljom će se za I=8 izračunati sledeće vrednosti:

Y(8)

Y(13)

Y(18)

Y(23)

.

.

.

Y(98)

Ovako organizovanom petljom će se za I=9 izračunati sledeće vrednosti:

Y(9)

Y(14)

Y(19)

Y(24)

.

.

.

Y(99)

Ovako organizovanom petljom će se za I=10 izračunati sledeće vrednosti:

Y(10)

Y(15)

Y(20)

Y(25)

.

.

.

Y(100)

Dakle, pet puta (za I=6 do 10) će se izvršavati 20 naredbi. Ovo nam ukazuje na ideju da možemo predstaviti sve ovu putem dve ugnježdene petlje.

NPR:

```
For  (I=6; I<=10; I++)
     For      (J=0; J<=90; J+=5)
        Y(I+J)=Y(I+J-5)+Y(I+J);
```

Da proverimo da li smo ovom transformacijom izbegli loop Carry zavisnost.

Za I=6 i J=0 imamo:

Y(6)=Y(1)+Y(6)

Za I=6 i J=5 imamo:

Y(11)=Y(6)+Y(11)

Izgleda da smo se prešli, zato što i dalje imamo loop carry zavisnost. Međutim, iz naigled bezizlazne situacije postoji vrlo lak izlaz. Naime, ako samo jednostavno zamenimo mesta brojačima I i J onda ćemo dobiti sledeću petlju (par ugnježdenih petlji).

```
For  (I=0; I<=90; I+=5)
     For      (J=6; J<=10; J++)
         Y(I+J)=Y(I+J-5)+Y(I+J);
```

Ovoga puta, zavisnosti nema :)

Za J=0 i I=6

Y(6)=Y(1)+Y(6)

ZaJ=0 i I=7

Y(7)=Y(2)+Y(7)

I tako dalje....

Dakle, prostom zamenom mesta brojača rešili smo problem.

Ukoliko bi sada hteli da ove dve petlje napišemo u paralelnom obliku to bi izgledalo ovako:

```
FOR (J=0; J<=0-; I+=5)
Y(J+6:10)=Y(J+1:5) + Y(J+6:10)
```

Zadatak br. 4

Korišćenjem GCD testa ispitati da li postoje Loop Carry zavisnosti u sledećoj petlji:

A)

```
FOR (I=1; I<=100; I++)
X(2*I+3)=X(2*I)+5)
```

B)

```
FOR (I=2; I<=100; I+=2)
A(I)=A(50*I+1);
```

C)

```
FOR (I=2; I<=100; I+=2)
A(I)=A(I-1);
```

Rešenje zadatka br. 4

Po GCD testu u petlji postoji Loop Carry zavisnost ako u svakoj dimenziji naredbe NZD(A,C) celobrojno deli D–B, pri čemu je oblik indeksa sa leve strane A_I + B, a sa desne C_I + D. Da bi se GCD primenio petlja mora da bude normalizovana. Normalizacija podrazumeva da se petlja inkrementira uvek za ++1 a ne za neku proizvoljnu vrednost.

A)

NZD(A,C)=2

D–B=0–3=–3

Dakle, (–3):2 => ne postji Loop Carry zavisnost.

B)

*Normalizacija:

```
For (I=1; I<=50; I++)
a(2*I)=a(100*I+1)
```

A=2

B=0

C=100

D=1

NZD(A,C)=2

D–B=0–1=–1

Dakle, (–1):2 => ne postji Loop Carry zavisnost.

C)

For(I=1. I<=50,I++)

a(2*I)=a(2*I–1)

A=2,

B=0,

C=2,

D=–1

=> NZD(A,C)=2 D–B=–1 =>nema loop carry zavisnosti.

GCD test je dobar da kaže da nema navisnosti. Tj. kada on kaže da nema, tada ih sigurno nema, ali ako on kaže da zavisnosti ima, onda postoji verovatnoća da zavisnosti zaista nema jer on ne uzima u obzir granice same petlje. Ovu anomaliju ćemo ilustrovati sledećim primerom.

For (I=2; I<=100, I+=2)

X(50*i+1)=X(i–1)*K

Ovu petlju prvo treba normalizovati: Dakle,

For (I=1; I<=50; I++)

X(100*+1) = X(2*I–2)*K;

A=100

B=1

C=2

D=–1

NZD(100,2)=2

D–B=–2

Dakle, po GCD testu Loop Carry zavisnost postoji.

Hajde sada to da proverimo:

Za I=1 X(101)=(1)*K

Za I=2 X(201)=X(3)*K

Za I=50 X(5001)=X(99)*k

Dakle, nema Loop Carry zavisnosti. Iz ovoga bi mogli da zaključimo da GCD test ne valje. Međutuim, zaboravili smo da GCD test ne uzmima u obzir granice petlje. Šta bi se desilo kada bi se petlja još samo jedared izvrtela.

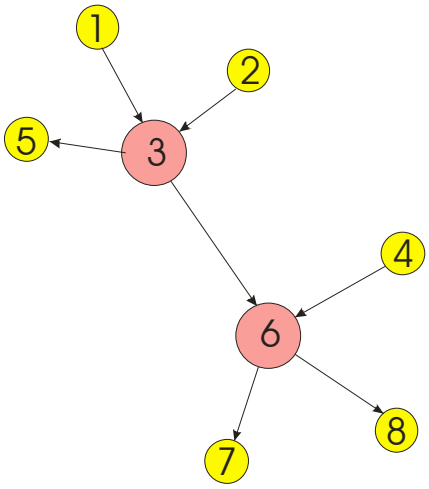
Za I=51 X(5101)=X(101)*K

Eto nama zavisnosti, znači, GCD test je ipak OK, s tim što ne uzima u obzir granice petlje.

Zadatak br. 5

Za sledeću sekvencu nardbi napisati paralelni program. Pretpostaviti da se u jednom zadatku paralelnog programa može izvršiti jedna ili više naredbi iz sekvence. Sinhronizaciju između zadataka izvršiti korišćenjem **semafora**.

1	A:=(F+C)/7;
2	B:=G+F-C;
3	C:=A*B;
4	D:=G+F;
5	A:=F-2*C;
6	IF (2G>C) THEN
	E:=B/2+D
	ELSE
	E:=C-F;
	END IF
7	D:=G+2F+4;
8	B:=(G-2F)*C



Rešenje zadatka br. 5

RAW	WAR	WAW	Ukupno
1-3	1-3	1-5	1-3
2-3	2-3	2-8	2-3
2-6	3-5	4-7	3-5
3-5	3-8		3-6
3-6	6-7		4-7
4-6	6-8		6-7
3-8			6-8

RAW hazardi nastaju između promenljive sa LEVE strane i iste te promenljive sa DESNE strane znaka jednakosti i to u naredni ispod posmatrane. (Read After Write) NPR: naredba broj 3 će pokušati da pročita podatak A koji je generisan u naredbi 1, tako da ove dve naredbe ne bi mogle da se vektorijzuju.

WAR hazardi nastaju između ukoliko postoji zavisnost podatka sa desne strane i istog tog podatka sa leve strane iz neke od narednih instrukcija.

WAW hazardi nastaju ukoliko postoji zavisnost ukoliko se u različitim naredbama dodele dodeljuje vrednost istom podatku.

Ovakva analiza omogućila nam je da sastavimo tabelu zavisnosti tj. konflikata koje ćemo morati da rešimo uz pomoć semafora.

```
Program Paralelizacija;
Var         A,B,C,D,E,F,G:real;      //Ovo su promenljive koje će se koristiti u programu.
            S_1_3: Semaphore         //Za svaku konfliktnu situaciju formiramo po jedan monitor
            S_2_3: Semaphore         //
            S_3_5: Semaphore         //
            S_3_6: Semaphore         //
            S_4_6: Semaphore         //
            S_6_7: Semaphore         //
            S_6_8: Semaphore         //

Procedure P_1;      //Procedura P_1 ima za cilj da obavi izračunavanje A:=(F+C)/7 i nakon toga da
Begin               //setuje semafor S_1_3 na vrednost V jer će sa linijom 3 nastati konfliktna situacija
A:=(f+c)/7;         //kada se to ne bi uradilo. Naime, setovanjem semafora na V praktično se postiže
V(S_1_3);           //paljenje crvenog svetla.
End;

Procedure P_2;      //Procedura P_2 radi isto to, samo računa svoje izračunavanje i setuje svoj semafor
Begin
B:=g+f-c;
V(S_2_3);
End;

Procedure P_3;      //Primetili smo da su i P_1 i P_2 pored svojih izračunavanja palili crveno svetlo
Begin               //na semaforu koji se odnosi na konflikt između te naredbe i treće naredbe
P(S_1_3);           // ti semafori su S_1_3 i S_2_3
P(S_2_3);           //
C:=A*B;             //Sada, te semafore moramo prebaciti na zeleno svetlo setovanjem na vrednost P,
V(S_3_5);           //nakon toga moramo obaviti izračunavanje iz treće linije, i na kraju,
V(S_3_6);           //podesiti  crveno svetlo na svim semaforima koji se bave konfliktom
End;                //prouzrokovanim trećom naredbom (S_3_5 i S_3_6)
```

```
Procedure P_4;      //Proceduri P_4 niko do sada nije podesio crveno svetlo, tako da ona
Begin               // to ne mora da radi, ali zato ona mora pored svog izračunavanja da podesi
D:=g+f;             // crveno svetlo svim procedurama koje zavise od nje: dakle, S_4_6
V(S_4_6);
End;

//Ovakvim načinom razmišljanja napišu se i sve ostale procedure.
Procedure P_5;
Begin
P(S_3_5);
A:=f-2*c;
End;

Procedure P_6;
Begin
P(S_3_6);
P(S_3_6);
If (2g>c) then E:=b/(2+d) else
E:=c-f;
V(S_6_7);
V(S_6_8);
End;

Procedure P_7;
Begin
P(S_6_7);
D:=g+2f+4;
End;

Procedure P_8;
Begin
P(S_6_8);
B:=(g-2f)*c
End;
```

// Sve što sada ostaje da se uradi je da se ovi procesi puste u paralelno izvršavanje naredbom //COBEGIN.

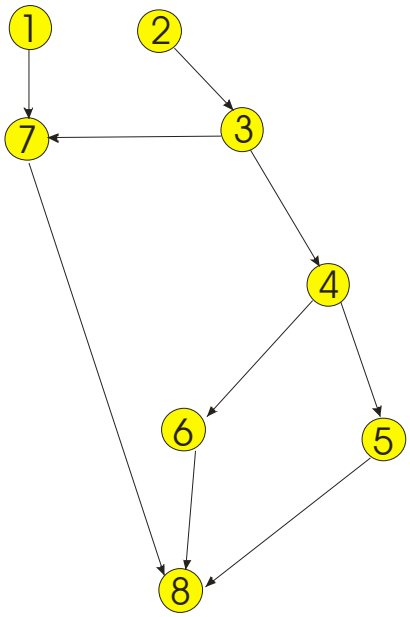
```
Begin
COBEGIN
P_1;
P_2;
P_3;
P_4;
P_5;
P_6;
P_7;
P_8;
COEND
End.
```

Zadatak br. 5a

Za sledeću sekvencu naredbi napisati paralelni program. Pretpostaviti da se u jednom zadatku paralelnog programa može izvršiti jedna ili više naredbi iz sekvence. Sinhronizaciju između zadataka izvršiti pomoću semafora.

Rešenje zadatka br. 5a

1	$A_8:=A_5+A_6+A_7$
2	IF (A_1>0) THEN
	$A_2:=A_3+A_4$
	ELSE
	$A_2:=A_3+A_5$
	END IF
3	$A_1=:(A_2+A_3)/A_4$
4	$A_2:=(A_3+A_4)/A_5$
5	$A_3:=(A_4+A_5)/A_6$
6	$A_2:=(A_5+A_7)/A_7$
7	$A_1:=(A_6+A_7)/A_8$
8	IF (A_1>0) THEN
	$A_2:=A_3+A_4$
	ELSE
	$A_2:=A_3+ A_5$
	END IF



RAW	WAR	WAW	Ukupno
1–7	2–3	2–4	1–7
2–3	2–5	2–6	2–3
3–8	2–7	2–8	3–4
5–8	3–4	3–7	3–7
7–8	3–5	4–6	4–5
	3–6	4–8	4–6
	3–8	6–8	5–8
	4–5		6–8
			7–8

RAW hazardi nastaju između promenljive sa LEVE strane i iste te promenljive sa DESNE strane znaka jednakosti i to u naredni ispod posmatrane. (Read After Write) NPR: naredba broj 3 će pokušati da pročita podatak A koji je generisan u naredbi 1, tako da ove dve naredbe ne bi mogle da se vektorijzuju.

WAR hazardi nastaju između promenljivih ukoliko postoji zavisnost podatka sa desne strane i istog tog podatka sa leve strane iz neke od narednih instrukcija.

WAW hazardi nastaju ukoliko postoji zavisnost ukoliko se u različitim naredbama dodele dodeljuje vrednost istom podatku.

Ovakva analiza omogućila nam je da sastavimo tabelu zavisnosti tj. konflikata koje ćemo morati da rešimo uz pomoću semafora.

```
Program Sekvenca;

Var A1,A2,A3,A4,A5,A6,A7,A8:Real;

S_1_7, S_2_3, S_3_4, S_3_7, S_4_5, S_4_6, S_5_8, S_6_8, S_7_8:Semaphore


Procedure P1;

Begin

    A8=A5+A6+A7;

    V(S_1_7);       //Pali se crveno svetlo na semaforu S_1_7

End;


Procedure P2;

Begin

    If A1>0 then A2:=A3+A4

    Else

    A2:=A3+A5

    End if

    V(S_2_3);       //Pali se crveno svetlo na semaforu S_2_3

End;


Procedure P3;

Begin

    P(S_2_3);       //Pali se zeleno svetlo na semaforu S_2_3

    A1:=(A2+A3)/A4;

    V(S_3_4);

    V(S_3_7);

End;


Procedure P4;

Begin

    P(S_3_4)              //Pali se zeleno svetlo na svim semaforima koji se završavaju na  4

    A2:=(A3+A4)/A5  // a crveno se pali na svim semaforima koji počinju na 4

    V(S_4_5);       //To je opšti princip
```

```
        V(S_4_6);
End;
```

```
Procedure P5;
Begin
     P(S_4_5)
     A3:=(A4+A5)/A6;
     V(S_5_8);
End;

Procedure P6;
Begin
     P(S_4_6)
     A2:=(A5+A6)/A7;
     V(S_6_8);
End;

Procedure P7;
Begin
     P(S_1_7);
     P(S_3_7);
     A1:=(A6+A7)/A3;
     V(S_7_8);
End;

Procedure P8;
Begin
     P(S_5_8);
     P(S_6_8);
     P(S_7_8);
     If A1>0 then A1:=A3+A4
     Else
     A2:=A3+A5
     End if
End;
```

```
//Glavni program
Begin
COBEGIN
     P1;
     P2;
     P3;
     P4;
     P5;
     P6;
     P7;
     P8;
COEND
End;
```

Zadatak br. 6

Rešiti problem međusobnog isključivanja n procesa uz pomoć **semafora**.

Rešenje zadatka br. 6

```
Program semafori;

Const       n=10;

Var         S:Semaphore;


Procedure Process(I:integer);

Begin

    Repeat

            P(S)

            Kritični deo;

            V(S);

    Forever;

End;


Begin

S:=1;

COBEGIN

    Proces(1);

    Proces(2);

    .

    .

    Proces(n);

COEND;

End.
```

Zadatak br. 7

Napisati program za rešenje problema ograničenog bafer. Ovaj problem se javlja kada više procesa upisuje poruke u bafer a više procesa čita iz bafera.

Svi procesi su konkurentni a u jednom trenutku samo jedan proces sme pristupiti baferu. Bafer je ograničene veličine.

Rešenje zadatka br. 7

Neka bafer može da primi N poruka.

Uvodimo 3 **semafora**:

- **Kapija** (binarni semafor koji treba da obezbedi uzajamno isključivanje pri pristupu baferu)
- **Puno** (opšti semafor koji treba da obezbedi da ne može da se čita iz praznog bafera i odgovara broju poruka u baferu)
- **Prazno** (opšti semafor koji treba da obezbedi da ne može da se upisuje u pun bafer i odgovara broju praznih mesta u baferu)

```
Program Ograničeni Bafer;
Var         Bafer:array(0..N–1) of Message; //U bafer će moći da stane N poruka.
            Pointer:0..N–1;  //Služi da ukazuje na poziciju elementa koji se uzima iz bafera.
            Count:0..N       //Broj popunjenih mesta u baferu
            Kapija, Puno, Prazno:Semaphore;       //Definišu se semafori !

Procedure proizvođač;              //Procedura koja će proizvoditi podatke koji se smeštaju u bafer.
Var         Stavka:message;
(generisanje poruke)
Begin                              //najpre se izgeneriše poruka, pa se onda kreće beskoačna
Repeat                             //Petlja koja ima za cilj sledeće...
     P(Prazno);                    //Na početju setuje semafor prazno na zeleno svetlo!!
     P(kapija);                    //Takođe, setuje i semafor kapija na zeleno svetlo
            Bafer ((pointer+Count) mod N) := Stavka
            Count:=Count+1;
     V(kapija);                    //Posle generisanja podatka vraća se crveno svetlo na semafor
     V(Puno);                             //kapija, i setuje se crveno svetlo na semaforu puno jer
Forever;                           //sada bafer nije prazan!!!!!
End;
```

```
Procedure potrošač;
Var  Stavka:message;
Begin
Repeat
     P(Puno);              //stavlja zeleno svetlo na semafor Puno, što znači da može da se
     P(Kapija);            //pročita poruka iz bafera. Naravno otvara se i kapija.
     Stavka :=Bafer(Pointer);
     Count=Count–1;
     Pointer = (pointer +1) mod N;
     V(kapija);            //Posle čitanja iz bafera, kapija se zatvara i podešava se da
     V(prazno);            //semafor prazno bude na crvenom svetlu.
(korišćenje poruke)
forever;
end;

Begin
     Puno:=0;
     Pointer:=0;
     Prazno:=N;
     Counter:=0
     Kapija=1;

COBEGIN
     Proizvođač;
     Potrošač;
COEND
End.
```

Zadatak br. 8

Rešiti problem proizvođač–potrošač u sistemu sa ograničenim baferom koristeći MONITOR.

Rešenje zadatka br. 8

```
Program Monitor;

Const       Kap=10;         //kapacitet bafera

Monitor ogran_bafer

Var         Bafer:array(0..kap–1) of integer;

            In,Out:Integer;

            N:integer;

            NotEmpty, NotFull:Condition;


Procedure Upisi (V:Integer)

Begin

    If N=kap then wait (NotFull);   //Ako je ispunjen kapacitet bafera onda se čeka na Condition

    B(IN):=V;                       //Na IN port stavlja vrednost V, koju upisujemo.

    IN := (IN+1) mod kap;           //In se uvećava za 1

    N=N+1;                          //Broj elemenata u baferu se takođe povećava

    Signal(NotEmpty);               //Šalje se signal da bafer nije prazan.

End;


Procedure Citaj(Var V:integer)

Begin

    If n=0 then wait(notEmpty);     //Ako je bafer prazan, čeka se na signal da nije prazan

    V=B(OUT);                       //U promenljivu V uzima se vrednost sa OUT porta

    OUT=(OUT+1)mod kap;             //Osvežava se tekuĆa vrednost OUT–a

    N=N–1;                          //Smanjuje se broj elementa u baferu

    Signal(NotFull);                //Šalje se signal da bafer više nije pun !!!!

End;

//Sada sledi inicijalizacija monitora !!!!

Begin (telo monitora)

    IN:=0           // ukazuje mesto gde se upisuje

    OUT:=0          //Ukazuje na mesto odakle se čita

    N:=0            //Broj zauzetih mesta u baferu

End;
```

(kraj monitora)

//Sada treba napisati procedure sa beskonačnim petljama koje će oponašati procese potrošnje i proizvodnje nekih promenljivih koje se čitaju ili upisuju u bafer.

```
Procedure Proizvođač
Var V:integer;
Begin
        Repeat
        Proizvodi(V);
        Ogran_bafer.upisi(V);
        Forever;
End;

Procedure Potrošač
Var     V:intweger;
Begin
        Repeat
        Ogran_bafer.citaj(V);
        Koristi.(V);
        Forever;
End;

Begin (glavni program)
COBEGIN
        Proizvođač;
        Potrošač;
COEND;
End.
```

Iz prethodnog primera se vidi da je potrebno izvršiti sinhronizaciju tj. slanje signala. To se radi uvođenjem promnljivih tipa CONDITION. Nad pomenljivim tipa CONDITION u monitoru se mogu izvršiti dve operacije:

- WAIT(C) Proces u kome se ova naredba izvršava se blokira tj. stavlja se u red čekanja koji odgovara uslovu C

- Signal(C) ako red čekanja za uslov C nije prazan aktivirati prvi proces u redu čekanja. Ako je red prazan naredba nema dejstvo.

Ove naredbe se ne smeju mešati sa P i V u semaforima. Za razliku od semafora za promenljive tipa CONDITION nije bitna brojna vrednost. Brojne vrednosti se prenose i pamte pomoću izlaznih promenljivih u monitoru. Ovo je moguće jer se u jednom ternutku u monitoru nalazi samo jedan proces. **NAPOMENA**: Ako u monitoru postoji jedna promenljiva tipa CONDITION tada se nad monitorom formiraju dva reda čekanja: Red čekanja procesa koji čekaju da uđu u monitor i red čekanja procesa koji čekaju na uslov.

Zadatak br. 9

Rešiti problem međusobnog isključivanja pomoću semafora, pri tome te semafore simulirati preko monitora.

Rešenje zadatka br. 9

Mogućnost simulacije pokazuje da se prelaskom sa semafora na monitore ne gubi ništa od funkcionalnosti!

```
Problem Simulacija;

Monitor Simulacija;

Var          S:0..1;

             NotBusy:Condition;

Procedure P;

Begin

     If S=0 then Wait (notbusy);

     S:=0;

End;


Procedure V;

Begin

     S:=1;

     Signal(NotBusy);

End;


Begin (monitora)

S=1;

End;


Procedure Proc1;

Begin

Repeat

     Simulacija.P;

     Kritični1;

     Simulacija.V;

     Rem1;

Forever
```

```
End;
```

```
Procedure Proc2;
Begin
Repeat
     Simulacija.P;
     Kritični2;
     Simulacija.V;
     Rem2;
Forever
End;

Begin
COBEGIN
     Proc1;
     Proc2;
COEND;
End.
```

Zadatak br. 10

Procesi A i B generišu podatke za procese C,D i E i smeštaju ih u redove čekanja R1 i R2 iste dužine N. proces A naizmenično upisuje po jedan podatak u redove R1 i R2, a proces B upisuje naizmenično po dva podatka u R1 i jedan podatak u R2. Proces C čita Po jedan Podatak samo iz reda R1, a proces D čita po jedan podatak samo iz reda R2. Proces E čita naizmenično po jedan podatak iz reda R1, pa jedan podatak iz reda R2.

Ostvariti sinhronizaciju i komunikaciju pomoću monitora, tako da različiti procesi mogu istovremeno da pristupaju različitim redovima.

Rešenje zadatka br. 10

```
Program Procesi;

Type podatak = record

....

end;

Const       DUZ=N;


Monitor Red_1;

Var         R1:array (0..N-1) of podatak;

            Point_R1:0..N-1;

            Count_R1:0..N

            NotFull_R1, NotEmpty:Condition;


Procedure Upisi_R1(X:podatak)

Begin

    If (Count_R1=N) then wait (NotFull_R1);

    R1 (Point_R1+Count_R1) mod N) := X;

    Count_R1 := Count_R1 + 1;

    Signal (NotEmpty_R1);

End;


Procedure Citaj_R1 (X:podatak);

Begin

    If (Count_R1=0) then wait (notempty_R1);

    X:=R1(Point_R1);

    Count_R1 :=Count_R1 - 1;

    Signal (NotFull_R1);
```

```
End;
```

```
Begin
Count_R1 :=0;
Point_R1 :=0;
End;

Monitor Red_2;
Var R2:Array(0..N-1) of Podatak;
Point_R2:0..N-1;
Count_R2:0..N;
NotFull_R2, NotEmpty_R2:Condition;

Procedure Upisi_R2(X:podatak)
Begin
    If (Count_R2=N) then wait (NotFull_R2);
    R2 (Point_R2+Count_R2) mod N) := X;
    Count_R2 := Count_R2 + 1;
    Signal (NotEmpty_R2);
End;

Procedure Citaj_R2 (X:podatak);
Begin
    If (Count_R2=0) then wait (notempty_R2);
    X:=R2(Point_R2);
    Count_R2 :=Count_R2 - 1;
    Signal (NotFull_R2);
End;

Begin
    Count_R2:=0;
    Point_R2:=0;
End;

Procedure: P_A;
Begin
    Repeat
```

```
        Generiše(X);
        Red_1.Upisi_R1(X);
        Generise(X);
        Red_2.Upisi_R2(X)
        Forever
End;

Procedure: P_B;
Begin
        Repeat
        Generiše(X);
        Red_1.Upisi_R1(X);
        Generise(X);
        Red_1.Upisi_R1(X);
        Generise(X);
        Red_2.Upisi_R2(X)
        Forever
End;

Procedure: P_C;
Begin
        Repeat
        Red_1.Citaj_R1(X);
        Koristi(X);
        Forever
End;

Procedure: P_D;
Begin
        Repeat
        Red_2.Citaj_R1(X);
        Koristi(X);
        Forever
End;
```

```
Procedure: P_E;
Begin
     Repeat
     Red_1.Citaj_R1(X);
     Koristi(X);
     Red_2.Citaj_R2(X);
     Koristi(X);
     Forever
End;

Begin
     COBEGIN
     P_A;
     P_B;
     P_C;
     P_D;
     P_E;
     COEND
End;
```

Zadatak br. 11

Rešiti problem međusobnog isključivanja dva procesa u pristupu zajedničkom resursu pomoću semafora. Semafor simulirati nitima u Javi.

Rešenje Zadatka br. 11

```
Import java.lang.threed;

Public Class Semafor_Nit;

Protected Int S;

Public Semafor_Nit

( S:=1;)

Public Synchronized void P()

( While (S==0)

Try (Wait ( );) catch (Interupted Exception ex) ( );

S=0;

)


Public Synchronized void V( )

( If (S==0)

Notify( );

S=1;

)


Public Class Proc1 implements java.lang.runable

Protected Semafor_Niti A;

Public proc1 (Semafor_Niti B)

(A=B;)

Public void Run( )

( for ( i ; )

(A.P;

krit.sek1

A.V;

Ostatak;

)

)
```

```
Public Class Proc2 implements java.lang.runable
Protected Semafor_Niti A;
Public proc2 (Semafor_Niti B)
(A=B;)
Public void Run( )
( for ( i ; )
(A.P();
krit.sek2
A.V();
Ostatak;
)
)
```

```
Public class primer
Public Static void main (string argv( ))
(Semafor_niti c=new semafor_niti() ;
proc1 PR1=new proc1 (c)
proc2 PR2=new proc2 (c)
new Thread(Pr1).Start;
new Thread(Pr2).Start;
)
)
```

Zadatak br. 12

Procesi A i B generišu podatke za proces C i smeštaju ih u redove čekanja R1 i R2 dužine N. Proces A naizmenino upisuje po jedan podatak u R1 i po jedan u R2. Proces B upisuje podatke samo u red R2. Proces C naizmenično čita podatke iz redova R1 i R2. Prilikom čitanja iz reda R1 proces C čita samo jedan podatak, a prilikom čitanja iz reda R2 proces C čita jedan podatak ako je u ovaj red zadnji upisivao proces B, ili dva podatka ako je u ovaj red zadnji upisivao proces A.

Sinhronizovati pomoću monitora.

Rešenje zadatka br. 12

```
Program Procesi;

Type podatak = record

....

end;

Const       DUZ=N;


Monitor Red_1;

Var         R1:array (0..N–1) of podatak;

            Point_R1:0..N–1;

            Count_R1:0..N

            NotFull_R1, NotEmpty:Condition;


Procedure Upisi_R1(X:podatak)

Begin

    If (Count_R1=N) then wait (NotFull_R1);

    R1 (Point_R1+Count_R1) mod N) := X;

    Count_R1 := Count_R1 + 1;

    Poslednji := Poslednji;

    Signal (NotEmpty_R1);

End;


Procedure Citaj_R1 (X:podatak);

Begin

    If (Count_R1=0) then wait (notempty_R1);

    X:=R1(Point_R1);

    Count_R1 :=Count_R1 – 1;

    Pointer_R1 := (Pointer_R1 + 1) mod N;
```

```
    Signal (NotFull_R1);
End;
```

```
Begin
Count_R1 :=0;
Point_R1 :=0;
End;

Monitor Red_2;
Var R2:Array(0..N-1) of Podatak;
Point_R2:0..N-1;
Count_R2:0..N;
PoslednjiUpisao: 0..1;        //0 znači da je poslednji bio A a jedinica da je poslednji bio B
NotFull_R2, NotEmpty_R2:Condition;

Procedure Upisi_R2(X:podatak, Posl:0..1)
Begin
     If (Count_R2=N) then wait (NotFull_R2);
     R2 (Point_R2+Count_R2) mod N) := X;
     Count_R2 := Count_R2 + 1;
     Poslednji:=Posl;
Signal (NotEmpty_R2);
End;

Procedure Citaj_R2 (X:podatak);
Begin
     If (Count_R2=0) then wait (notempty_R2);
     X:=R2(Point_R2);
     Count_R2 :=Count_R2 - 1;
     Pointer_R2 := (Pointer_R2 + 1) mod N;
     Signal (NotFull_R2);
End;

Begin
     Count_R2:=0;
     Point_R2:=0;
End;
```

// Posle ovoga monitor za red2 vodi računa i o tome koji proces je zadnji upisao u taj red, zato što od toga zavisi koliko će se podataka pročitati iz reda broj 2. Sve što sada teba uraditi je da se napišu procesi koji se ponašaju na opisan način.

```
Procedure: P_A;
Begin
     Repeat
     Generiše(X);
     Red_1.Upisi_R1(X);
     Generise(X);
     Red_2.Upisi_R2(X,0)     //Ova nula služi za evidenciju koji proces je upisao poslednji u red 2
     Forever                 //U ovom slučaju to je proces A (nula označava proces A).
End;

Procedure: P_B;
Begin
     Repeat
     Generise(X);
     Red_2.Upisi_R2(X,1)     //Ova jedinica služi za evidenciju koji proces je upisao poslednji u red
2
     Forever                 //U ovom slučaju to je proces B (nula označava proces B).
End;

Procedure: P_C;
Begin
     Repeat
     Red_1.Citaj_R1(X)
     Koristi(X);

     If Red_2.Poslednji = 0 Then
     Red_2.Citaj_R2(X)
     Koristi(X);
     Red_2.Citaj_R2(X)
     Koristi(X);
     Else
     Red_2.Citaj_R2(X)
     Koristi(X);
     Forever
End;
```

```
// Ovim smo obezbedili da se struktura ponaša kao u opisu zadatka zar ne ;)
// Ostaje asmo još da napišemo glavni program

Begin
     COBEGIN
     P_A;
     P_B;
     P_C;
     COEND
End;
```

ZADATAK BR. 13

Proces A generiše podatke za procese B i C i smešta ih u M različitih redova R1, R2, R3, ...RM od kojih je svaki dužine N. Procesi B i C čitaju podatke iz redova tako što red iz koga čitaju podatak biraju slučajno, funkcijom Random(M) koja generiše slučajan broj u opsegu 1..M.

Ako je proces A upravo upisao podatak u red Ri sledeći generisani podatak se upisuje u Ri+1 pri čemu se preskače red iz koga je proces B zadnji put čitao podatak.

Napisati program na ADI kojim se realizuje opisana struktura, pri čemu treba obezbediti istovremeni pristup različitim redovima.

Rešenje zadatka br. 13

```
Procedure Zredovi is

Type Podatak is ... //Neki zapis ili slog;


Task type Red is

    Entry Upis (X:In Podatak);

    Entry Citanje(X:Out Podatak);

End red;


M:const Integer :=10;

SubType RedIndex:Integer range 1..M

Redovi:array(1..M) of Red;

Task PA;

Task PC;

Task PB is

    Entry IndeksZadnjegReda (I:Out Integer);

End PB;


Task Body Red is

N:=Count Integer:=20;

R:array(0..N-1) of Podatak;

Point,Count: Integer:=0;

Begin

    Loop

        Select

            When (Count < N) =>

                Accept Upis (X:In podatak) do
```

```
                End upis;
                Count := Count+1;
        Or
        When (Count >=0) =>
        Accept Citanje (X:Out Podatak) do
        X := R(Point);
        End Citanje;
        Count :=Count –1;
        Point :=(Point+1) mod N;
    End Select
  End Loop
End Red;

Task Body PA is
X:podatak
I,J:redIndex;
Begin
  I:=1;
  While (I<=M) do
    Loop
      PB.IndeksZadnjegReda(J);
      If (I<>J) then
      Generisi(X);
      Redovi(I).Upisi(X);
      I:=I+1;
      End if;
    End loop
  End loop
End PA;
NIJE DOVRšENO !!!!!!!!!
```

ZADATAK BR. 14

Napisati Program za rešavanje problema dva proizvođača i jedan potrošač. Procesi Proizvođač_1 i proizvođač_2 pristupaju baferu ograničene veličine radi upisa poruke, a proces Potrošač radi čitanja poruke. Istovremen pristup baferu nije moguć. Rešiti taj problem korišćenjem:

A) Semafora

B) Monitora

Rešenje zadatka br. 14

a)

```
Program 2x1;
Var bafer:array(0..N–1) of poruka;
Pointer:0..N–1;
Brojač:0..N;
S,Puno,Prazno:Semaphore;

Procedure Proizvođač_1;
Var Podatak:Poruka;
Begin
    Repeat
            Proizvodnja_poruka(podatak);
            P(Prazno);
            P(S);
            Bafer( (Pointer+Brojač) mod N) := Podatak;
            Brojač := Brojač+1;
            V(S);
            V(Puno);
    Forever;
End;

Procedure Proizvođač_2;
Var Podatak:Poruka;
Begin
    Repeat
            Proizvodnja_poruka(Podatak);
            P(Prazno);
            P(S)
```

```
            Bafer( (Pointer+Brojač) mod N) := Podatak
            Brojač := Brojač –1;
            V(S);
            V(Puno);
    Forever;
End;

Procedure Potrošač;
Var podatak:Poruka;
Begin
    Repeat
            P(Puno);
            P(S);
            Podatak := Bafer(Pointer);
            Brojač := brojač–1;
            Pointer :=(Pointer–1) Mod N;
            V(S);
            V(Prazno);
            Orišćenje_Poruke (Podatak);
    Forever;
End;

Begin
Puno := 0
Prazno := N;
S:= 1;
Brojač := 0;
Pointer: = 0;

COBEGIN
    Proizvođač1;
    Proizvođač2;
    Potrošač;
COEND;
End;
```

b)

```
Program 2x1;
Const Kapacitet = 100;
Monitor Ograničeni_Bafer;
Var Bafer:array(o..Kapacitet–1) of Poruka;
In,Out,N :Integer;
NotEmpty,NotFull:Condition;

Procedure Upisi(Podatak:poruka)
Begin
     If (N=Kapacitet) the wait (NotFull);
     Bafer(IN) :=Podatak;
     In :=(In+1) mod kapacitet;
     N:=N+1;
     Signal(NotEmpty);
End;

Procedure Citaj (Podatak:Poruka);
Begin
     If (N=0) the Wait(NotEmpty);
     Podatak := Bafer(Out);
     Out:=(Out+1) mod Kapacitet;
     N:=N–1;
     Signal (NotFull);
End;

Begin
     In:=0;
     Out:=0;
     N:=0;
End;
```

```
Procedure Proizvođač_1;
Var podatak:poruka;
Begin
     Repeat
            Proizvodnja_Poruke(Podatak);
            Ograničeni_Bafer.Upiši(podatak);
     Forever
End;

Procedure Proizvođač_2;
Var podatak:Poruka;
Begin
     Repeat
            Proizvodnja_Poruke(Podatak);
            Ograničeni_bafer.Upiši(Podatak);
     Forever;
End;

Ptrocedure Potrošač;
Var podatak:poruka;
Begin
     Repeat
            Ograničeni_bafer.Citaj(Podatak);
            Korišćenje_Poruke(Podatak);
            Forever
     End;

// Glavni Program

Begin
     COBEGIN
     Proizvođa_1;
     Proizvođa_2;
     Potrošač;
     COEND
```

```
End;
```

Zadatak br. 15

Proces A generiše podatke za procese B i C i smešta ih u redove čekanja R1 i R2 dužina M i N. Proces A upisuje podatak u onaj red čekanja koji u trenutku upisa sadrži manji broj podataka a ukoliko u oba reda trenutno postoji isti broj podataka upisuje u red R1. Proces B čita podatke samo iz reda R1, proces C samo iz reda R2. Napisati program na programskom jeziku ADA kojim se realizuje opisana struktura, pri čemu treba obzbediti istoveremeni pristup procesa B i C odgovarajućim redovima.

Rešenje zadatka br. 15

```
Procedure Baferovanje;

Type poruka is record ... end;


Task Bafer_1 is
     Entry Upis_Poruke(X: In_Poruka);
     Entry Citanje_Poruke(X: Out_Poruka);
     Entry Broj_Poruka(Br: Out Integer);
End Bafer_1;


Task Bafer_2 is
     Entry Upis_Poruke(X: In_Poruka);
     Entry Citanje_Poruke(X: Out_Poruka);
     Entry Broj_Poruka(Br: Out Integer);
End Bafer_2;


Task Proces_A;
Task Proces_B;
Task Proces_C;


// Sada se pišu tela taskova


Tak Body Bafer_1 is
M:Const Integer :=100;
R1:array(0..M-1) of poruka
Point, Count, Br:Integer:=0;
Begin
     Loop
```

```
            Select
            When (Count <M) =>
                    Accept Upis_Poruke(X:In Poruka) Do
                    R1(( Point – Count) mod M) := X
                    End Upis_Poruke;
                    Count :=Count+1;
            Or
                    When (Count >0) =>
                    Accept Citanje_Poruke  (X:Out Poruka) Do
                    X:=R1(Point)
                    End Citanje_Poruke;
                    Count := Count – 1
                    Point := (Point+1) Mod M;
            Or
                    Accept Broj_Poruka (Bp:Out Integer) Do
                    Br := Count
                    End Broj_Poruka;
            End Select;
    End Loop;
End Bafer_1;

Tak Body Bafer_2 is
N:Const Integer :=100;
R2:array(0..N–1) of poruka
Point, Count, Br:Integer:=0;
Begin
    Loop
            Select
            When (Count <N) =>
                    Accept Upis_Poruke(X:In Poruka) Do
                    R2(( Point – Count) mod M) := X
                    End Upis_Poruke;
                    Count :=Count+1;
            Or
                    When (Count >0) =>
```

```
                    Accept Citanje_Poruke  (X:Out Poruka) Do
                    X:=R2(Point)
                    End Citanje_Poruke;
                    Count := Count – 1
                    Point := (Point+1) Mod M;
            Or
                    Accept Broj_Poruka (Bp:Out Integer) Do
                    Br := Count
                    End Broj_Poruka;
            End Select;
    End Loop;
End Bafer_2;

Task Body ProcesA is
Procedure Generisanje_Poruke(X:Out Poruka) Is
End Generisanje_poruke;
Bp1,Bp2:Integer;
X:Poruka;
Begin
    Loop
    Generisanje_Poruke(X);
    Bp1:=Bafer_1.Broj_Poruka(Bp1);
    Bp2:=Bafer_2.Broj_Poruka(Bp2);
    If (Bp1>Bp2) =>
    Bafer_2.Upis_Poruke(x);
    Else
    Bafer_1.Upis_Poruke(x);
    End Loop;
End Proces_A;

Task Body Ptroces_B is
Procedure Korišćenje_Poruke (X:In Poruka) Is
End korišćenje_Poruke;
X: Poruka;
Begin
```

```
    Loop
    Bafer_1.Citanje_Poruke(X);
    Korišćenje_Poruke(X);
    End loop;
End Proces_B;


Task Body Ptroces_C is
Procedure Korišćenje_Poruke (X:In Poruka) Is
End korišćenje_Poruke;
X: Poruka;
Begin
    Loop
    Bafer_2.Citanje_Poruke(X);
    Korišćenje_Poruke(X);
    End loop;
End Proces_C;

//Glavni Program !!!!
BEGIN
NULL
END BAFEROVANJE
```

Zadatak br. 16

Procesi A i B generišu podatke za proces C i smeštaju ih u redove R1,R2,R3....RM dužina N. Proces A upisuje podatke tako što u redove sa parnim indeksom upisuje po jedan podatak, dok u redove sa neparnim indeksom po dva podatka. Proces B upisuje podatke redom u sve redove. Prilikom čitanja, proces C čita podatak iz reda Ri ako je u taj red zadnji upisivao proces A ili ga preskače ako je zadnju upisivao proces B.

Simulirati takvu strutturu na ADI!

Rešenje zadatka br. 16

```
Procedure Redovi

Poruka is record ... End record;

Task Type Red Is

    Entry Upis (X: In Poruka, P: In Integer);

    Entry Citanje (X: Out Poruka);

    ENTRY BrojPoruka (Bp:out integer);

    Entry BrojProcesa(PBr: out Integer);


M: Const Integer :=10;

R:Array(0..M-1) of Red;


Task A;

Task B;

Task C;


Task Body Red is

N:Const Integer :=100;

Bafer:Array(O..N-1) of Poruka;

Point, Count :Integer := 0;

Proces:Integer;

Begin

    Loop

        Select

            When (Count<N) =>

            Accept Upis(X: in Poruka, P: In Integer) do

            Bafer (( Point - Count( Mod N) := X;

            Proces :=P;

            End Upis;
```

```
                Count := Count +1;
                Point := (Point – 1) mod N;
        Or
                When (Count > 0) =>
                Accept Citanje (X: Out Poruka) do
                X:= Bafer(Point);
                End Citanje;
                Count := Count–1;
        Or
                Accept BrojPoruka(Bp: Integer) do
                Bp:=Count;
                End BrojPoruka;
        Or
                Accept BrojProcesa (PBp: Out Integer) do
                PBp := Proces;
                End BrojProcesa;
        End select
    End Loop
End Red;

Task Body A is
Procedure generisanje_Poruke (X: Out Poruka) is
End generisanje_Poruke;
I: Integer :=0;
X:Poruka;
Begin
    Loop
        Generisanje_Poruke(X);
        If (( I mod 2) = 0) =>
                R(I).Upis (X,1);
        Else
                Begin
                R(I).Upis( X,1);
                R(I).Upis( X,1);
                End;
```

```
            I:=(I+1) Mod M;
    End Loop;
End A;
Task Body B is
Procdure Generisanje_Poruke (X: Out Poruka) is
End Generisanje_Poruke;
X:Poruka;
I:Integer :=0;
Begin
    Loop
            GenerisanjePoruke(X,2);
            I:=(I+1) Mod M;
    End Loop;
End B;

Task Body C is
Procedure KorišćenjePoruke( X: in Integer) is
End KorišćenjePoruke;
X:Poruka;
I,RedniBrojProcesa:Integer :=0;
Begin
    Loop
            R(i).BrojProcesa(RedniBrojProcesa);
            If (RedniBrojProcesa) =1) =>
            Begin
                    R(i).Citanje(X);
                    KorišćenjePoruke(X);
            End;
            I:=(I+1) Mod M;
    End Loop;
End C;



//GLAVNI PROGRAM
```

```
BEGIN

NULL;

END REDOVI
```

Zadatak br. 17

Rešenje zadatka br. 17

ZADACI (ACTUS)

Actus je paralelni programski jezik zasnovan na paskalu. Osnovna komponenta jezika ACTUS iz koje izviru sve osobine koje ga karakterišu kao paralelni jezik je tip podataka. Paralelizam je ugrađen u tip polje ili niz. Tako se u ACTUS–u definišu dve vrste polja:

- Skalarna
- Vektorska

U **skalarnim** poljima je elementima pristupa jednom po jednom elementu polja, a kod **vektorskih** polja se elementima pristupa paralelno tj. istovremeno u cilju vršenja paralelne obrade.

Primer definisanja polja:

```
Const N=100;
Var  XPAR: array (1:N) of real;
     YSEQ: array (1..N) of real;
```

XPAR je paralelno polje, a YSEQ je sekvencijalno (obično) polje.

Što se tiče vektorskih polja, moramo napomenuti jedno bitno ograničenje. Naime, vektorsko polje može biti samo jednodimenzionalno tj. ukoliko mora da bude višedimenzionalno samo jedna dimenzija može biti vektorska.

Primer:

```
Const      N=200
           M=100;
Var        S:array (1:M,1..N) of real;
```

Ovim delom ACTUS koda je definisano vektorsko polje čijim se **kolonama** može pristupati parallno, a **vrstama** se mora pristupati sekvencionalno. Dakle, kada se navodi specifikacija tj. deklarisanje polja onda se prvo definišu kolone, a zatim vrste u matrici.

Paralelna konstanta:

Paralelna konstanta mora da se slaže po dimenziji za vektorskim poljem sa kojim se nalazi u istoj operaciji.

Ona takođe uzima sukcesivne vrednosti iz definicije (1:N).

Za nju se može definisati i inkrement kojimože biti i pozitivan i negtivan.

Konkatenacija konstanti se dobija njihovim navođenjem u definiciji pri čemu su one razdovojene zarezom.

NPR:

```
Parconst PC=–2:(4)14, 22:(–3)1 ,–17:(6)91;
```

Ako primenimo sve ovo što smo rečima definisali, onda možemo analizirati vrednost PC paralelne konstante. Dakle,

PC=–2,2,6,10,14 ,22,19,17,15,13,11,9,7,5,3,1 ,–17,–11,–5,1,7,....,91

Indeksni skup:

Indeksni skup je vrsta paralelne konstante koja se koristi za indeksiranje paralelnih promenljivih. U definicijama se navodi kao poslednja iza svih konstanti i promenljivih.

Primer:

```
Var     a:array(1:20) of integer;
Index   Set1=2:15;
        Set2=6:18;
```

Nad indeksnim skupom se mogu vršiti razne operacije, kao što su:

- Unija A(Set1+Set2) –> A(2:18)
- Presek A(Set1*Set2) –> A(6:15)
- Razlika A(Set1–Set2) –> A(2:5)
- Shift
- Rotate

Ovi operatori se primenjuju kod indeksiranja paralelnih promenljivih i format u kome se javljaju je:

Opseg paralelizma operator celobrojni izraz

Prilikom određivanja kojim elementima paralelnog polja se pristupa, indeksima koji definišu opseg paralelizma dodaje se celobrojni izraz. Rezultujući indeksi moraju biti u oblasti definisanoj za to polje

Primer:

```
Var     A,B,C:array(1:100) of real;
Index   control=4:25+30:40;
Begin
C(control) := A(control) + B(control shift –1)
```

Ovaj programčić obavlja ovo:

```
C(4) := A(4) + B(3)
.
.
.
C(40) := A(40) + B(39)
```

Dakle, šiftovan je indeksni set za jednu poziciju u levo.

Kada bi umesto shift –1 stavili rotate –1 onda bi rezultat izgledao ovako:

C(4) := A(4) + B(3)

.

.

.

C(40) := A(40) + B(1)

WITHIN naredba:

Ovu naredbu najbolje možete upoznati na primeru:

```
Within M:N do
Begin
    A(#):=B(#)+C(#);
    B(#):=B(#) + 1;
End;
```

Ovaj program ima za posledicu to da izvrši dve naredbe koje su date između begin i end za sve vrednosti # od M do N. Dakle, sama reč nam kaže **within** a to znači unutar (misli se unutar datog opsega).

Uslovne naredbe IF i CASE

Uslovne naredbe koje manipulišu vektorskim poljima u ACTUS–u su definisane kao i odgovarajuće naredbe u paskalu.

IF naredba: Za if naredbu postoje dva slučajua:

1: kada se u uslovu ne javljaju vektorska polja, tj. paralelne promenljive. Tada se u uslovu javljaju skalarne promenljive ili elementi skalarnih polja. Stepen paralelizma, za paralelne promenljive koje se javljaju u granama IF naredbe, treba biti eksplicitno definisani, na primer:

```
If J>0 then
A(1:N) := A(1:N)/2.0
```

Ili

```
Within 2:49 do
Begin
A(#):=–B(#);
If J>0 then A(#) := A(#) +3.0
End;
```

2: Druga varijanta je kada se u uslovu javljaju paralelne promenljive. Stepen paralelizma promenljivih u granama **nesme** biti eksplicitno definisan, odnosno mora biti posredno definisan preko simbola #, an primer:

```
IF A(1:N) > 0.0 then
A(#):=–A(#)
```

U ovom slučaju se prvo paralelno izračuna uslov, za svaku komponentu A(1:N) posebno, pa se posle naredba A(#):=–A(#) izvrši samo za one elemente vektorskog polja koji su zadovoljili traženi uslov.

Ili

```
Within 1:P do
Begin
B(#) :=1.0;
If A(#)>0.0 then
A(#):=A(#)–1;
X(#):=A(#)
End;
```

Ako pak postoji ELSE grana onda se ona izvršava za one elemente vektorskog polja koji ne zadovoljavaju uslov.

Ako su na obe strane u uslovu paralelne promenljive, stepen paralelizma treba biti jednak za obe strane, a uslov se ispituje za svaki element posebno. U then grani se opet selektivno izvršavaju naredbe samo za one elemente za koje je uslov bio ispunjen, na primer:

```
IF A(1:3) > B(1:3) then
Behin
    A(#):=0;
    B(#):=0;
End;
```

Osim prethodno opisane IF naredbe moguća su i sledeća proširenja naredbe:

- A: IF ANY (test) then S

Ova naredba ima značenje da će se naredba S izvršiti nad svim elementima paralelnih promenljivih čak i ako makar jedan elment iz paralelne promenljive zadovoljava uslov (test).

- B: IF ALL (test) then S

Ova naredba će se izvršiti nad svim elementima paralelne promenljive ako i samo ako svi elementi zadovoljavaju uslov (test)

CASE naredba:

Što se tiče CASE naredbe, za nju važi da se za svaki element paralelne promenljive iz uslova izvršava odgovarajuća naredba nad tim elementom, kao na primer:

```
TYPE        zemljište=(okean, led, zemlja, pustinja);
Var         površina:array(1:50) of zemljište;

.
.
.
CASE površina (15:50) of
Okean:      naredba_1;
Led,pustinja: naredba_2;
Zemlja:     Naredba_3;
End;
```

Naredbe petlji

To su naredbe WHILE, REPEAT i FOR. Naravno, i ove naredbe su nasleđene iz paskala.

While:

WHILE (uslov) DO naredba;

U uslovu za ovu naredbu mogu se javiti skalarne i vektorske promenljive. Ako se u uslovu javi skalarna promenljiva stepen paralelizma za paralelne promenljive koje se javljaju u telu petlje treba da bude eksplicitno naveden nao na primer:

```
While i>0 do
Begin
A(1:10) :=A(1:10)*2;
I:=i–1
End;
```

Ako se pak u uslovu naredbe javi vektorska promenljiva, telo petlje se izvršava samo za one elemente paralelne promenljive nad kojima su vršene neke operacije u iteraciji i. Na primer:

```
While A(1:N) > 0 then
```

```
A(#) := A(#) – 1;
```

Da bi se lakše razumelo ono što smo malopre definisali napisaću tabelu koja predstavlja tok izvršenja programa. Dakle,

1. iteracija	A(1)=4	A(2)=2	A(5)=6
2. iteracija	A(1)=3	A(2)=1	A(5)=5
3.	A(1)=2	A(2)=0	A(5)=4
4.	A(1)=1		A(5)=3
5.	A(1)=0		A(5)=2
6.			A(5)=1
7.			A(5)=0

Kao i kod IF naredbe i ovde se mogu koristiti operatori ANY i ALL.

Repeat naredba:

Repeat naredba until (uslov);

Pošto se kod ove naredbe telo petlje nalazi ispred uslova, stepen paralelizma u telu petlje ne može biti promenljiv kao kod WHILE naredbe. Na priemer:

```
Var          A,B:array(1:100) of integer;
             I,J:1..100;
.
.
repeat
     A(1:N) := B(1:N);
     I:=I+1;
     J:=J–1;
     Within 1:N – I:J Do        //Minus ovde označava razliku skupova
     If B(#) <0 then
     B(#) :=B(#)+1;
Until ALL (A(1:N)>0);
```

For naredba:

```
FOR control:=START (BY increment) To FINISH DO S;

FOR control:=START (BY decrement) DownTo FINISH DO S;
```

```
ParaConst    DIAGONALA=1:100;
             ID:array(1:100) of integer;
Within 1:100 do
     For ID(1:100) := DIAGONALA to 100 DO
     A(ID(#)) = B(ID(#))
```

Potprogrami

Postoje dve vrste potpograma, to su Funkcije i potprogrami. Funkcije mogu vratiti jednu ili više vrednosti zavisno od toga da li se radi o skalarnoj ili vektorskoj strukturi. Ulazni parametri funkcije mogu biti skalarnog ili vektorskog tipa i njihov stepen paralelizma ne mora biti isti sa stepenom paralelizma rezultata.

Na primer:

```
Type         VectorA=array(1:50) of real;
             VectorB=array(1:100) of real;
Var          A:VectorA;
             B:VectorB;
             Z:real;
Function     CALCULATE(N:real, Y:VectorB):vectorA;
Const        P=200
             Q=50
             P1=199;
Var          ZZ:array(1:P,1:Q) of real;
Index        Inside=1:P1
Begin
..
.
CALCULATE := ZZ(1:50,1)
.
..
End;
```

Postoji nekoliko standardnih funkcija koje operišu nad svim elementima paralelne promenljive a kao rezultat daju skalarnu vrednost. Na primer:

FIRST(X(#))	Vraća index prvog člana vektorskog polja
SUM(X(#))	Vraća sumu svih elemenata vektorskog polja

Na primer:

Izračunati prosečnu vrednost elemenata vektora X koji su pozitivni!

```
Const       N=100;
Var         A,X:array(1:N) of real;
            Average:Real;
Begin
            A(1:N):=0.0;
            If X(1:N) >0.0 then
            Begin
            A(#) :=1;
            Average := SUM(X(#)) / Sum(A(#));
            End;
End.
```

Procedure!

```
Type        Vector=array(1:100) of real;
            Matrix=array(1:50,1..10) of integer;

.
Procedure DAX(S,T:Matrix, var P:Real, Var U,V:Vector);
Begin
.
Telo procedure
.
End;


Begin

...
...
...
DAX(X,Y,R,A,B)
..
```

end;

Moguće je i preneti niz sa promenljivim opsegom paralelizma

NPR:

Procedure P(var A:array(EXTENT) of real);

Poziv ovakve procedure bi bio:

P(B(1:10))

P(B(I:J))

P(B(#))

P(B(ODDS)), gde je ODDS već definisan indeksni skup neparnih brojeva.

Zadatak br. 1

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Napisati program na ACTUS–u za izračunavanje sume elemenata u svakoj vrsti matrice MAT dimezija 15x100.

Rešenje zadatka br. 1

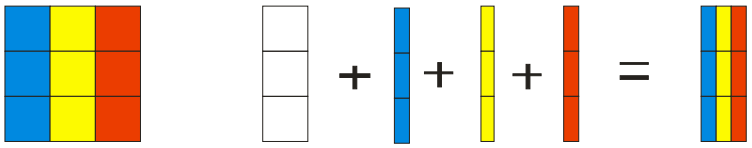
```
Program Suma_vrsta

Const       N=15;
            M=100;
Var         MAT:array(1:N,1..M) of real;        :array (kolone, vrste)
            I: integer;
            SUM:array (1:N) of real;
Begin
            Sum(1:N):=0
            For I:=1 to M
            Begin
            SUM (1:N) := SUM (1:N) + MAT (1:N,I);
            End;
End.
```

Diskusija:

Na početku se kao konstante definišu granice (dimenzije) matrice. Ovo nije obavezan korak, ali je preporučljivo zbog toga što kada bi morali da promenimo dimenzije matrice koju obrađujemo, morali bi da menjamo ceo kod (čak i deo za izračunavanje). Ali, ako dimenzije predstavimo kao konstante, onda nema takvih problema, jer je sve što je potrebno uraditi promena vrednosti konstante na početku programa.

Zatim, ide odeljak posvećen deklarisanju promenljivih. Moramo da definišemo promenljivu MAT, koja će biti vektorska promenljiva, i to tako da joj vektorska dimenzija bude dimenzija **kolone**. Ovde ste se verovatno zbunili, zašto nam je vektorska dimenzija dimezija kolona kada nama treba zbir svih vrsta. Trik je u tome, što ćemo zbir svih vrsta dobiti tako što ćemo sabrati svaku kolonu (dakle, vrste paralelno) sa promeljivom SUM i time dobiti sume svih vrsti. Za lakše razumevanje pogledajte ilustraciju:



Dakle, praznoj sumi (vekoru) sabiramo prvu kolonu, pa zatim drugu kolonu pa zatim treću kolonu. I na kraju dobijemo u prvom elementu promenljive SUM zbir svih elementata prve vrste (što smo i tražili), u drugom ćemo imati isto to samo za druge elemente i tako dalje. Dakle, poenta je u tome što se u isto vreme dobija zbir svih vrsta, a to se postiže ako se kolone (nižu) sabiraju jedna za drugom.

Upravo to radi i program.

Brojač prolazi kroz petlju onoliko puta koliko ima kolona (M puta), za svaki prolaz se sumi (koja je inicijalno prazna) dodaju se (vektorski) kompletne kolone i time se polako formira ono što smo tražili.

Zadatak br. 2

Napisati program na ACTUS–u za množenje vektora matricom, čiji su svi elementi ispod i na glavnoj dijagonali jednaki jedinici, a iznad glavne dijagonale jednaki nuli.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_1 + x_2 \\ x_1 + x_2 + x_3 \\ x_1 + x_2 + x_3 + x_4 \end{bmatrix}$$

Rešenje zadatka br. 2

```
Program Množenje;

Const       N=15;

Var         MAT:array(1:N,1..N) of real;

            I: 1..N;

            X:array(1..N) of real;

            Y:array(1:N) of real;

Begin

    //Inicijalizacija matrice

    MAT(1:N,1) := 1.0;

    For I:=2 to N do

    Begin

            MAT(I:N, I) := 1.0;

            MAT(1:I–1, I):=0.0;

    End;

    //Inicijalizacija vektora

    Y(1:N) := 0.0;


    //računanje proizvoda

    For I:=1 to N do

            Y(1:N) := Y(1:N) + MAT(1:N,I)*X(I)

End.
```

Diskusija:

Ovo je kvadratna matrica tj. N=M. Da se malo podsetimo kako se beše množi matrica sa vektorom!

Množenje matrice sa vektorom se postiže tako što se pomnoži prvi element vektora sa prvim elementom prve vrste, zatim se tom proizvodu **doda (sabere)** proizvod prvog elementa vektora i drugog elementa prve vrste i tako daje dok se ne uradi i poslednji element prve vrste.

Sada prelazimo na drugi element u vektoru i na drugu vrstu i ponovimo ceo postupak. Ovaj postupak se vrši za sve članove vektora. Dakle, na osnovu tog algoritma radi program.

Odmah posle begin naredbe mora se izvršiti inicijalizacija matrica. To ćemo uraditi tako što ćemo svim PRVIM elementima svih vrsta dodeliti vrednost 1.0 Dakle, naredbom MAT(1:N,1) := 1.0; smo postigli sledeći efekat:

$$\begin{bmatrix} 1 & \\ 1 & \\ 1 & \\ 1 & \end{bmatrix}$$

sada treba ostatak matrice inicijalizovati onako kako je dato u zadatku.

Pošto smo sve prve članove svih vrsti setovali, ostaje nam da setujemo sve ostale tj. 0d 2 do N. Zato puštamo petlju da ide od 2 do N.

Naredba MAT(I:N, I) := 1.0; (naravno uzimajući u obzir da se nalazi u petlji tj. izvršava se nekoliko put) setovaće na jedinicu u prvom prolazu kroz petlju sve elemente od drugog do N–tog u drugoj koloni na jedinicu. Zatim će setovati sve elemente od trećeg do N–tog u trećoj koloni i tako treba nastaviti do kraja.

Naravno, primećujemo da nam takva matrica ne treba, pa moramo izvršiti korekciju time što ćemo setovati na nulu (u prvom prolazu) sve elemente od prvog do prvog u prvoj koloni, od prvog do drugog u drugoj koloni, od prvog do trećeg u trećoj koloni dakle, od prvog do I–tog u I–toj koloni.

To ćemo uraditi naredbom

MAT(1:I–1, I):=0.0;

Posle ovoga imamo inicijalizovanu matricu M onako kako nam treba.

Sada možemo početi da primenjujemo pravilo množenja koje smo malopre objasnili (obnovili).

Na početku setujemo vektor Y na nulu. Zatim pustimopetlju da mota od 1 do N. Primećujemo da je dimenzija kolona u vektoru Y vektorska. To znači da ćemo rezultat dobijati tako što ćemo paralelno određivati sve elemente ciljnog niza Y.

Imajući u vidu da se petlja mota od I=1 do I=N taj posao će nam obaviti naredba:

Y(1:N) := Y(1:N) + MAT(1:N,I)*X(i)

Zadatak br. 3

Napisati program na ACTUS–u za formiranje matrice $A_{100x100}$ tako da je a_{ij}=i+j.

Rešenje zadatka br. 3

```
Program matrica

Const       N=100;

ParConst    NIZ = 1:N;

Var         MAT:array(1:N,1..N) of integer;

            J:1..N;

Begin

    For J:=1 to N do

            MAT(1:N,J) := NIZ + J;

End;
```

Diskusija:

Definišemo konstantu N i dodeljujemo joj vrednost 100. Paralelna konstanta NIZ dobija vrednosti od 1 do N (redom).

Matrica se klasično definiše (ne inicijalizuje se). J je brojač za petlju.

Vrtimo petlju da bi smo prošli kroz sve kolone. Tj. u svakom krugu se bavimo različitom kolonom. Dakle, u prvom krugu se dodeljuje elementima 1:N,1 vrednost NIZ + J. Dakle, Prva kolona dobija vrednosti:

$$\begin{bmatrix} 1+1 \\ 2+1 \\ 3+1 \\ 4+1 \end{bmatrix}$$

U drugom krugu se dobija:

$$\begin{bmatrix} 1+1 & 1+2 \\ 2+1 & 2+2 \\ 3+1 & 3+2 \\ 4+1 & 4+2 \end{bmatrix}$$

Na kraju se dobija:

$$\begin{bmatrix} 1+1 & 1+2 & 1+3 & 1+4 \\ 2+1 & 2+2 & 2+3 & 2+4 \\ 3+1 & 3+2 & 3+3 & 3+4 \\ 4+1 & 4+2 & 4+3 & 4+4 \end{bmatrix}$$

Zaključujemo da smo to upravo i hteli da dobjemo. Znači, zadatak je OK

Zadatak br. 4

Napisati program na ACTUS–u za izračunavanje ciklične konvolucije dva niza X(100) i Y(100). Ciklična konvolucija se definiše izrazom:

$$Z(K)=\sum_{i=0}^{99} X(i) * y((i+k)\bmod 100) \qquad \text{k=0,1,2,3,4,.....99}$$

Rešenje zadatka br. 4

Izgleda jako komplikovano zar ne ?

Pokušaćemo da provalimo zavisnost u iteracijama tj. kako se menjaju indeksi.

Za K=0 Važi:

X_0 Y_0

X_1 Y_1

X_2 Y_2

X_3 Y_3

Zbir proizvoda svih ovih parova daće konvoluciju.

Za K=1 Važi:

X_0 Y_1

X_1 Y_2

X_2 Y_3

X_3 Y_0

Za K=2 Važi:

X_0 Y_2

X_1 Y_3

X_2 Y_0

X_3 Y_1

Za K=3 Važi:

X_0 Y_3

X_1 Y_0

X_2 Y_1

X_3 Y_2

Dakle, možemo da zaključimo da se indeksi rotiraju ulevo !!!

Ovo zapažanje ću iskoristiti u programu za mehanizam promene indeksa koji indeksiraju nizove X i Y.

```
Program Ciklična_konvolucija
Const       N=100;
Var         X:array(0..N–1) of integer;
            Y,Z:array(0:N–1) of integer;
            I:Integer
Index       Ind:0:N–1;
Begin
            Z(ind):=0;
            For I:=0 to N–1 Do
    Begin
            Z(ind):=Z(ind)+y(ind)*X(i)
            Y(ind) :=Y(ind rotate 1);
    End
End.
```

Diskusija:

Definišu se konstanta N, pomenljive X,Y,Z i I. Takođe se definiše indeksna promenljiva koja indeksira elemente od nultog do (N–1)–og.

Na početku programa se Z niz (koji je ustvari rezultujići niz našeg programa) setuje na 0. Zatim se pokrene petlja koja će se motati od 0 do N–1. U telu ove petlje će se računati niz Z, tako što će se nizu Z iz prošlog trenutka dodavati proizvod celog niza Y sa elementom X(I). Posle savke ovakve obavljene radenje, treba rotirati vektor (niz) Y po istom zakonu koji smo provalili u prvom delu zadatka (analizi problema).

Ovim je program završen.

Zadatak br. 5

Napisati program na ACTUSU za množenje 2 binarna broja $X_{n-1}X_{n-2}....X_1X_0$ i $Y_{n-1}Y_{n-2}.....Y_1Y_0$ i to u dve varijante:

a) bez korišćenja naredbe within

b) sa korišćenjem naredbe within

Brojeve X i Y predstaviti kao nizove slučajno inicijalizovanih logičkih (1 ili 0) verdnosti.

Rešenje zadatka br. 5

Prvo se moramo podsetiti opšteg principa množenja dva binarna broja:

Sledeća tablica prikazuje kako se množe dva 4–bitna broja

A_3	A_2	A_1	A_0	X	B_3	B_2	B_1	B_0	=	A_3B_0	A_2B_0	A_1B_0	A_0B_0
									A_3B_1	A_2B_1	A_2B_1	A_0B_1	
								A_3B_2	A_2B_2	A_1B_2	A_0B_2		
							A_3B_3	A_2B_3	A_1B_3	A_0B_3			
						P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

U našem slučaju mi ćemo množiti brojeve X i Y a proizvod ćemo označiti sa Z. Dakle, analizirajući ovu tablicu, možemo definisati pavila na osnovu kojih ožemo da generalizujemo množenje bilo koja dva binarna broja.

Dakle, možemo napisati sledeću tablicu:

Z(i)	XY	CY(J–1)	CY(J)	Z(i)
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

U zelenom delu tablice nalaze se sve moguće kombinacije tekućeg režultata (Z(I)), proizvoda XY i prenosa i predhodne iteracije. Na osnovu ovoga, ručno smo izračunali ostatak tablice u cilju da utvrdimo zavisnost među bitovima.

$$CY(J) = \overline{Z(i)} \cdot XY \cdot CY(j-1) + Z(i) \cdot \overline{XY} \cdot CY(j-1) + Z(i) \cdot XY \cdot \overline{CY(j-1)} +$$
$$+ Z(i) \cdot XY \cdot CY(j-1) + CY(j-1) \cdot (Z(i) \oplus XY) + Z(i) \cdot XY$$

Posle malo sređivanja dobija se:

$$Z(i) = \overline{Z(i)} \cdot \overline{XY} \cdot CY(j-1) + \overline{Z(i)} \cdot XY \cdot \overline{CY(j-1)} + Z(i) \cdot \overline{XY} \cdot \overline{CY(j-1)} + Z(i) \cdot XY \cdot CY(j-1)$$
$$= Z(i) \cdot (XY \oplus CY(j-1)) + Z(i) \cdot (XY \oplus CY(j-1))$$
$$= Z(i) \oplus XY \oplus CY(j-1)$$

Na osnovu dobijene zavisnosti među bitovima, možemo vrlo lako napisati ACTUS program:

```
Program_množenje_BEZ_WITHIN_naredbe
Const       N=4;
Var         Y:array(0..N–1) of boolean;
            X:array(0:N–1) of boolean;
            C,Cp:array(0:N–1) of boolean;
            Z:array(0:2*N–1) of boolean;
            I:0..N;
Index       p=0:N–1;
            Q=0:2*N–1;
Begin
    For i:=1 to N–1 do
    begin
    Randomize;
            If (random(2)=0 then
                    Y(i):=false
                    Else
                    Y(i):=true;
            Endif
    End;
            Z(Q):=false;
            Cp(P):=false;
            For i:=0 to N–1 do
    Begin
            C(P):=((Z(p shift i) XOR (X(P) and Y(i)) and Cp(P)) or Z(P shift i) and (X(P) and Y(i))
```

```
        Z(P shift i) := Z)P shift i) XOR (X(P) and Y(i)) XOR Cp(P);
        Cp(P) := C(P);
End;
        Z(2*N–1:2*N–1) := Cp(N–1:N–1)
```

```
Program_množenje_SA_WITHIN_naredbom
Const       N=4;
Var         Y:array(0..N–1) of boolean;
            X:array(0:N–1) of boolean;
            C,Cp:array(0:N–1) of boolean;
            Z:array(0:2*N–1) of boolean;
            I:0..N;
Index       p=0:N–1;
            Q=0:2*N–1;
Begin
For i:=0 to N–1 do
    Begin
    Within P do
            Begin
            C(#):=Z(#) XOR (X(#) and Y(i)) and Cp(#)) or Z(#) and (X(#) and Y(i))
            Z(#) :=Z(#) XOR (X(#) and Y(i)) XOR Cp(#);
            Cp(#):=C(#);
    End;
    Z(Q) := Z(Q rotate 1)
    End;
Z(Q) := Z(Q rotate N)
Z(2*N–1) := Cp(N–1:N–1);
End.
```

Zadatak br. 6

Koristeći nezavisno indeksiranje napisati program na ACTUS–u koji postavlja 0 (nule) na glavnoj i sporenoj dijagonali matrice $A_{50\times50}$

Nezavisno indeksiranje je pristup proizvoljnim elementima polja i odnosi se na višedimezionalna polja. Ovo se postiže tako što se kao indeks za sekvencijalnu koordinatu uzima paralelna promenljiva.

Rešenje zadatka br. 6

```
Program Matrica_sa_nezavisnim_indeksiranjem

Const       N=50

ParConst    GLDIAG=1:N;

            SPDIAG=N(–1)1;

Var         A:array(1:N,1..N) of integer;

            I:array(1:N) of integer;

Begin

            I:=GLDIAG;

            A(1:N,I):=0;

            I:=SPDIAG;

            A(1:N,I) :=0;

End.
```

Diskusija:

Najpre se definišu: konstanta N, paralelne konstante GLDIAG i SPDIAG i potrebne promenljive. Paralelna konstanta GLDIAG je konstanta vektorskog oblika koja se kreće od 1 do 50, a paralelna konstanta SPDIAG se kreće od 50 do 1 sa korakom –1.

Ovo nas navodi na razmišljanje: Šta će biti ove konstante u toku izvršenja programa. Logično je pretpostaviti da glavna dijagonala ima elemente koji su indeksiraju kao: prvi element prve vrste, drugi element druge vrste, treći element treće vrste i tako redom. Što se sporedne dijagonale tiče, njeni elementi indeksiraju pedeseti element prve vrste, četrdesetdeveti element druge vrste i tako dalje .... Upravo zbog toga elementi indeksnog skupa SPDIAG idu od M do 1 sa korakom –1.

Pored ovih konstanti, potrebene su nam i promenljive A i I. Promenljiva A je matrica, čija je dimenzija vrste paralelna, a promenljiva I je paralelni niz.

Na početku programa se paralelnom nizu / dodeljuje paralelna konstanta GLDIAG i izvrši se dodela vrednosti matrici A, tako da se sim vrstama (1:N) dodeli jedinica na odgovarajućem mestu označenim sa paralelnom konstantom I.

Isti postupak je i kod dodele vrednosti matrici kada indeksiramo sporednu dijagonalu.

Zadatak br. 7

Napisati program za transformisanje kvadratne matrice korišćenjem nezavisnog indeksiranja kao što je prikazano na slici.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix}$$

Ovo je tzv. transponovanje matrice tj. vrste i kolone menjaju mesta.

Rešenje zadatka br. 7

Da bi ovaj problem rešili, treba prvo utvrditi zakonitosti koje se javljaju u zadatoj transformaciji. Naime, primećijemo da se elementi na glavnoj dijagonali ne menjaju. Zatim, primećujemo da element A_{12} biva zamenjen elementom A_{21} znači indeksi su mu zamenili mesta. Ovo što smu uočili je sasvim dovoljno da se uradi program.

```
Program_Transponovnje
Const       N=20
ParConst    DIAG=1:20;
Var         A,ATR:array(1:N,1..N) of real;
            I,J:array(1:N) of integer
            K=0..N-1
Begin
            Within 1:N do
            Begin
                  I(#) := DIAG
                  J(#) := DIAG
                  For K:=0 to N-1 do
                  Begin
                        ATR(#,J(#)) := A(# rotate -k, I(#));
                        J(#) = J(# rotate -1);
                  End;
            End;
End.
```

Diskusija:

Prvo se definiše konstanta N koja će sadržati dimenziju matrice koja se transponije. Zatim ćemo definisati jednu paralelnu konstantu, koja će indeksirati elemente matrice na glavnoj dijagonali. Što se tiče promenljivih, definsaćemo matrice A i ATR sa dimenzijom vrste kao paralelnom dimenzijom i paralelne nizove I i J. Takođe, definišemo i brojač K.

Na samom početku programa Within naredba označava da će se telo ove naredbe (između begin i end) izvršiti za sve vrednosti simbola # od 1 do N.

Dakle, prvo se paralelnom nizu I dodeli vrednost parlelne konstante DIAG odnosno, I sada sadrži vrednost paralelne konstante DIAG. Isto to se uradi i sa paralelnim nizom J.

Zatim se pokrene petlja koja će motati od 0 do N–1. Formira se matrica ATR tako što se na njenim svim vrstama (tj. ovo označava prva dimenzija (#)) postavi vrsta matrice matrice A koja je rotirana za –K mesta, i to za??

Zadatak br. 8

Napisati program na ACTUS–u za množenje vektora X matricom koja je oblika $A_{nxn} \otimes I_S$ dakle, ovo je kronekerov proizvod matrice A_{nxn} i jedinične matrice I_s dimenzija sxs. Rešenje projektovati tako da se ne vrši množenje elementima matrice za koje se zna da su jdnaki nuli.

Primer: $\vec{Y} = (A_{3x3} \otimes I_3)\vec{X}$

$$\begin{bmatrix} a_{11} & 0 & 0 & a_{12} & 0 & 0 & a_{13} & 0 & 0 \\ 0 & a_{11} & 0 & 0 & a_{12} & 0 & 0 & a_{13} & 0 \\ 0 & 0 & a_{11} & 0 & 0 & a_{12} & 0 & 0 & a_{13} \\ a_{21} & 0 & 0 & a_{22} & 0 & 0 & a_{23} & 0 & 0 \\ 0 & a_{21} & 0 & 0 & a_{22} & 0 & 0 & a_{23} & 0 \\ 0 & 0 & a_{21} & 0 & 0 & a_{22} & 0 & 0 & a_{23} \\ a_{31} & 0 & 0 & a_{32} & 0 & 0 & a_{33} & 0 & 0 \\ 0 & a_{31} & 0 & 0 & a_{32} & 0 & 0 & a_{33} & 0 \\ 0 & 0 & a_{31} & 0 & 0 & a_{32} & 0 & 0 & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_4 + a_{13}x_7 \\ a_{11}x_2 + a_{12}x_5 + a_{13}x_8 \\ a_{11}x_3 + a_{12}x_6 + a_{13}x_9 \\ a_{21}x_1 + a_{22}x_4 + a_{23}x_7 \\ a_{21}x_2 + a_{22}x_5 + a_{23}x_8 \\ a_{21}x_3 + a_{22}x_6 + a_{23}x_9 \\ a_{31}x_1 + a_{32}x_4 + a_{33}x_7 \\ a_{31}x_2 + a_{32}x_5 + a_{33}x_8 \\ a_{31}x_3 + a_{32}x_6 + a_{33}x_9 \end{bmatrix}$$

Rešenje zadatka br. 8

```
Program Transformacija

Const       N=10;
            S=10;
Var         I,J:integer;
            A:array(1:N,1..N) of real;
            X:array(1..N*S) of real;
            Y:array(1:N*S) of real;
Begin
            Y(1:N*S) :=0;    //Na pocetku se rezultujuci vektor setuje na 0


            For J:=1 to S do
            For I:=1 to N do
            Y(J:(S)N*S) := Y(J:(S)N*S) + A(1:N,I) * X((I-1)*S + J)
End.
```

Diskusija:

Poenta je da se ne generiše matrica pomoću koje treba da se vrši množenje, nego da se utvrdi zakonitost u konačnom vektoru i da se taj vektor direkno generiše (bez množenja matrica).

Na početku se naš budući rezultujući vektor inicijalizuje nulama. Zatim se puste dve petlje da motaju. Jedna će motati od 1 do S, a druga od 1 do N.

Šta je ustvari ovo S?

S je dimenzija jedinične matrice koja se pominje u kronekerovom proizvodu. Iz primera za N=3 i S=3 zaključili smo da rezultujući vektor Y mora da ima dimenziju X*S.

Hajmo sada da vidimo šta ustvari rade ove dve petlje?!

Y(J:(S)N*S) := Y(J:(S)N*S) + A(1:N,I) * X((I-1)*S + J)

Vektoru Y na mestima indeksiranim indeksnim skupom od J do NS sa korakom S dodeljuje se zbir:

Elemenata na istim pozicijama rezultujućeg vektora Y ali iz prošle iteracije

Proizvoda elemenata matrice A koji su ustvari I–te kolone i vektora X indeksiranog sa (I–1)*S+J

Napomena:

Kada bi se umesto matrice A nalazila matrica B koja je oblika:

$$\begin{bmatrix} 0 & 0 & b_{11} & 0 & 0 & b_{12} & 0 & 0 & b_{13} \\ 0 & b_{11} & 0 & 0 & b_{12} & 0 & 0 & b_{13} & 0 \\ b_{11} & 0 & 0 & b_{12} & 0 & 0 & b_{13} & 0 & 0 \\ 0 & 0 & b_{21} & 0 & 0 & b_{22} & 0 & 0 & b_{23} \\ 0 & b_{21} & 0 & 0 & b_{22} & 0 & 0 & b_{23} & 0 \\ b_{21} & 0 & 0 & b_{22} & 0 & 0 & b_{23} & 0 & 0 \\ 0 & 0 & b_{31} & 0 & 0 & b_{32} & 0 & 0 & b_{33} \\ 0 & b_{31} & 0 & 0 & b_{32} & 0 & 0 & b_{33} & 0 \\ b_{31} & 0 & 0 & b_{32} & 0 & 0 & b_{33} & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} =$$

Onda bi se program razlikovao samo i jednoj liniji tj:

Y(J:(S)N*S) := Y(J:(S)N*S) + A(1:N,I) * X((I-1)*S + J)

Bi postalo

Y(J:(S)N*S) := Y(J:(S)N*S) + B(1:N,I) * X((I*S+1 - J)

Sve ostalo je isto!

Zadatak br. 9

Napisati paralelni program na ACTUS–u kojim se u slučajno inicijalizovanoj matrici X vrši:

- sortiranje **vrsta** u opadajući redosled (korišćenjem algoritma suseda),
- a zatim izvodi transponovanje novodobijene matrice nakon čega se dobija matrica XTR.
- Nakon toga se za svaku vrstu u matrici XTR pronalazi prosečna vrednost svih elemenata vrste koji su veći od nule..

Rešenje zadatka br. 9

```
Program Sortiranje_i_transponovanje

Const       M=100;
            N=200;

Var         XTR, X:array(1..M,1:N) of integer;      //M je dimenzija kolone, a N je dimenzija vrste
            Pom:array(1:N) of integer;              //POM je promenljiva koja ima dimenziju vrste
            I,J:integer;                            //I i J su brojačke promenljive
            Prosek:array(1:M) of real;      //ima dimenziju kolone, to je ustvari broj elemenata vrste

Index       Parni 2:(2)N–2;                         //Indeksira parne elemente u svakoj VRSTI
            Neparni 1:(2)N–1;                       //Indeksira neparne elemente u svakoj vrsti

Begin
            For J:=1 to N do                        //J će prolaziti kroz sve elemente svake vrste
            For I:=1 to M do                        //I će prolaziti kroz sve elemente svake kolone
            Begin
            Randomize;
            X(I, J:J) = Random(100);                // 1, 1:1 – adresira se prvi element prve kolone
                                                    // 2, 1:1 – adresira se prvi element druge kolone
                                                    // i tako dalje doks e ne inicijalizuje matrica
//Ovaj deo koda bio je zadužen da inicijalizuje matricu X random elementima
```

```
//Naredni deo koda služi za sortiranje (algoritmom suseda)
for I:=1 to M do                                    //I prolazi kroz sve kolone
     for J:=1 to (N div 2) do                       //J prolazi kroz vrste, ali samo do polovine
     Begin
     If X(i,Neparni) < X(i, neparni shift 1) then   //Ako su neparni elementi prve kolone < od
              Begin                                 //njima susednih elemenata onda će im se mesta
              Pom(#):=X(I,#);                       //zameniti
              X(i,#):=X(i,# shift 1);
              X(i, # shift 1) := pom(#);
              End;
     If X(i,parni) < X(i,parni shift 1) then        //Isto se radi i sa parnim elementima.
              Begin                                 //Nakom toga sve će biti sortirano
              Pom(#):=X(I,#);
              X(i,#):=X(i,# shift 1);
              X(i, # shift 1) := pom(#);
              End;
     End;

//Sledeći deo koda bavi se generisanjem transponovane matrice
P(#)=GLDIAG                                 //P(#) je paralelna konstanta za indeksni skup GLDIAG
For I:=1 to M do                            //I će da prolazi kroz sve kolone
XTR ( P(#), # rotate –I ) := X ( P(# rotate –I), #);    //MISIM DA OVAJ DEO KOA NIJE BAS U REDU!

//Sada se generiše matrica A koja mapira elemente XTR–a koji su veći od 0
For I:=1 to M do
Begin
              A(I,#) := 0;
              IF XTR (I,1:N) > 0 then
              A(I,#) := 1;
End;
//sada trebada se generiše niz sa prosečnim vrednostima vrsta
For I:=1 to M do
Begin
              If XTR (I,1:N) > 0 then
              Prosek(I) := sum (XTR(I,#)) / Sum(A(I,#) );
```

```
            End;

End.
```

Zadatak br. 10

Napisati program na ACTUS–u za sortiranje niza od P brojeva, koristeći algoritam suseda. Napomena: Kod ovog algoritma porede se elementi sa neparnim indeksima sa svojim desnim susedima i ako je sused manji oni zamene mesta. Posle toga se to uradi za elemente sa parnim endeksima. Ceo niz je uređen posle P/2 ovakvih koraka.

Rešenje zadatka br. 10

```
Program_algoritam suseda

Const       P=100;

Var         Pom, Niz:array (1:P) of integer;

            N:Integer;

Index       Neparni=1:(2)P–1;

            Parni=2:(2)P–2;

Begin

   For N=1 to (P div 2) do

      Begin

      If NIZ(neparni) > Niz(Neparni shift 1) then

            Begin

            POM(#):=NIZ(#)

            NIZ(#):=NIZ(# shift 1);

            NIZ(# shift 1):=POM(#)

            End;


      If NIZ(Parni) > Niz (parni shift 1) then

            Begin

            POM(#):=NIZ(#)

            NIZ(#):=NIZ(# shift 1);

            NIZ(# shift 1):=POM(#)

            End;

      End;

End.
```

Ovaj zadatak se rešavao na potpuno isti način kao i prethodni (što se sortiranja tiče).

Zadatak br. 12

Napisati program na ACTUS–u kojim se od celobrojne matrice A_{nxn} generiše matrica B u kojoj su zadržani elementi iz A ukoliko su prosti brojevi, dok su svi ostali elementi anulirani. Ukoliko su svi elementi nekih kolona matrice B prosti brojevi, elementima takvih kolona treba dodeliti vrednost N.

Elementi matrice A su slučajni brojevi, ali takvi da su različiti od nule i nisu veći od N.

Rešenje zadatka br. 12

```
Program prosti_brojevi;

Const       N=100;

Var         A,B:array(1:N,1..N) of Integer;     //Dimenzija kolone je paralelna
            L:array(1:N) of boolean;            //L je paralelna logička (boolean) promenljiva kapaciteta
            I,J,K:integer;                      //Brojači

Begin

            For J:=1 to N do                    //J će prolaziti kroz sve kolone
                  For I:= 1 to N do             //I će prolaziti kroz sve vrste
                        Begin
                        Randomize;
                        A(I:I,J) := random(N)+1   // 1:1,1 – prvi element prve kolone
                        End;                      // 2:2,1 – drugi element prve kolone
                                                  // I tako dok se kolona ne ispuni


//Sada treba analizirati dobijenu kolonu na proste brojeve, i rezultat te analize ćemo smestiti u vektor L
                  Within 1:N do
                        Begin
                              L(#) := False;
                              For K:=2 to N do
                              If (A (#,J) <> K  AND  (A (#,j) mod K = 0) then
                              L(#) := true
                        End;
//Analizom vektora L utvrđujemo kako ćemo generisati metricu B
                  If L(1:N) then B(#,J) :=0
                  Else B(#,J) := A(#,J);
                  If All (B(1:N,J) <> 0) then B(1:N,J) := N
End.
```

Zadatak br. 13

Napisati program na ACTUS–u kojim se na osnovu elemenata X(I,J) matrice X i elemenata Y(I,J) matrice Y formira matrica Z pri čemu su njeni elementi dobijeni na sledeći način:

$$Z(I,J)=\sum_{K=0}^{N-1} X(I,J)\cdot Y(I,K+J-1)\bmod N$$

Rešenje zadatka br. 13

```
Program Matrice;

Const       M=100;
            N=200;
Var         X:array(0..M–1, 0..N–1) of integer;
            I,J,K:integer;
Index       P=0:N–1;


Begin
    For I:=0 to M–1 Do
            Begin
                    For J:=0 to N–1 do
                    Begin
                    Randomize;
                    X(I,J:J) := random(1000);
                    Randomize;
                    Y(I,J:J) := random(1000);
                    End;
            Z(I,P) := 0;
                    For K:=0 to N–1 Do
                    Begin
                    Z(I,P) := Z(I,P)+X(I,P)*Y(I,P);
                    Y(I,P) := Y(I,P rotate 1)
                    End;
            End;
End.
```

Zadatak br. 14

Napisati program u ACTUS–u kojim se slučajno inicijalizovanoj matrici X_{2MxN} sortiraju elementi po kolonma u opadajući redosled. Za sortiranje koristiti algoritam suseda.

Rešenje zadatka br. 14

```
Program sortiranje

Const       M=25;
            N=100;
Var         X:array(1..2M,1..N) of integer;
            Pom:array(1:2*M) of integer;
            I,J:Integer;
Index       Neparni=1:(2) 2*M–1;
            Parni=2:(2) 2*M–2;
Begin
            For I:=1 to 2*M do
    Begin
            Randomize;
            X(I,I,J) := random(1000);
    End;
            For I:=1 to M do
    Begin
            If X(parni) > X(neparni shift 1,J) then
                    Begin
                    Pom(#) := X(#,J);
                    X(#,J) := X(# shift 1,J);
                    X(# shift 1,J) := pom(#);
                    End;
            If X(parni) > X(Parni shift 1,J) then
                    Begin
                    Pom(#) := X(#,J);
                    X(#,J) := X(shift 1, J);
                    X(# shift 1,J) := pom(#);
                    End;
End.
```

Zadatak br. 15

Napisati program na ACTUS–u za uređenje niza A od parnog broja elemenata N u neopadajući reosled metodom (potpuno mešanje zamena).

U programu možete koristiti SORT1(N,Var X) i SORT2(N,Var Y) kojima se pozivaju podrazumevani potprogrami za uređenje elemenata niza X i Y respektivno. N je ceo broj. X paralelna promenljiva sa opsegom paralelizma 1:N i Y paralelne promenljiva sa opsegom paralelizma 1:(2)2*N–1

Rešenje zadatka br. 15

```
Program Sortiranje;

Const       N=100;

Var         A,Pom:array(1:N) of integer;

            I:Integer;

Index       Prva_polovina = 1:N/2

            Neparni = 1:(2) 2*N–1

            Parni = 2:(2) 2N–2;

Begin

            For I:=1 to N do

            Begin

            Randomize;

            A(I,J):=random(1000);

            End;


            SORT1 ( N/2, A (Prva polovina) );

            Pom (Prva polovina) := A (prva polovina shift N/2);

            SORT1 (N/2, pom(prva polovina));

            A (prva polovina shift N/2) := pom (prva polovina);

            SORT2 (N/2, A (neparni));

            Pom (neparni) :=A (neparni shift 1);

            SORT2 (N/2, pom (neparni));

            A (neparni shift 1) := pom (neparni);


    If (A(parni) > a(parni shift 2) then

            Begin

            Pom(#) < A(#)

            A(#) := A(# shift 1);

            A(# shift 1) := Pom(#);
```

```
            End;
End.
```

Zadatak br. 16

Elementi binarne matrice A koji su iznad glavne i ispod sporedne dijagobnale imaju vrednost 1. Elementi matrice B su random elementi. Matrica C jednaka je proizvodu A*B. Napisati program na ACTUS–u u kome treba generisati samo matricu B, i na osnovu korišćenja njenih osobina generisati matricu C.

Rešenje zadatka br. 16

$$\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} \\ b_{51} & b_{52} & b_{53} & b_{54} & b_{55} \end{bmatrix} = \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}$$

```
Program množenje
Const       N=200;
            M=100;
Var         B:array (1..N,1..M) of integer;
            C:array (1:N,1..M) of integer;
            K,J: integer;
Begin
Randomize;
For I:= 1 to N do
For J=1 to M do
B(I,J)=random(100)    //Generisanje random matice

For J:=1 to N do
    Begin
    C(1:N,J) :=0;                  //Mota se jedna po jedna kolona.
    For I:=(N div 2) –2 to N do    //I mota od N/2–1 do N

    C ( (N–1–I):(I–1),J ) := C ( (N–2–I):(I–1),J) + B(I,J) )
    End;
End.
```

Zadatak br. 17

U matrici A_{nxn} elementi elementi za čije indekse važi I>J jednaki su Nuli, dok su u matrici B_{NxM} jednaki nuli elementi matrice za čije indekse važi I<J. Napisati ACTUS program za množenje AxB=C, pri čemu treba izbeći množenje elemenata čija je vrednost 0.

U programu predvideti generisanje matrice A i B. Takođe, u dobijenoj matrici C anulirati sve one kolone koje imaju bar neki element jednak nuli, a u ostalim kolonama ukoliko su svi elementi kolone veći od N, svim elementima takve kolone dodeliti upravo vrednost N.

Rešenje zadatka br. 17

```
Program Množenje_matrica

Const       N=100;
            M=200;
Var         A:array(1:N,I..N) of integer;
            B:array(1..N,1..M) of integer;
            C:array(1:N,1..M) of integer;
            I,J:integer;
Begin
For J=1 to N do
    Begin
            A(J+1:N,J) := 0;          //Defniše gde treba da budu nule!
            For I:=1 to J do          //Ova petlja postavlja ostale (nenulte elemente)
                    Begin
                    Randomize;
                    A(I,I:I) = random(100)
                    End;
    End;
//Potpuno isti postupak se ponovi za matricu B s tom razlikom što se na drugim mestima postavljau nule. Sve u svemu Princip je isti.
For J:=1 to N do
    Begin
            B(1:J-1,J):=0;
            For I=J to N do
                    Begin
                    Randomize;
                    B(I:I,J) := random(100)
                    End;
```

```
End;
```

```
 //Radi se kolona po kolona! Naime, J–ta kolona se najpre inicijalizuje pa se zatim formira konačan izgled na mestima na kojima ne treba da ostane nula.


For J:=1 to M do
     Begin
     C(1:N,J) := 0
              For I:=J to N do
              C(1:I,J) := C(1:I,J) + A(1:I,I)*B(I,J);


              // Zatim se primene poslednja dva usova zadatka i to je  to
              If Any C(1:N,J) >0 then
              C(1:N,J) = 0
              If All C(1:N,J) > N then
              C(1:N,J) = N;
     End;
End.
```

Zadatak br. 18

Zadate su realne matrice A_{NxN} i B_{NxM}. Na ACTUS–u napisati program kojim se od matrice A generiše njoj transponovana matrica C u odnosu na sporednu dijagonalu koa što je prikazano na primeru za N=4:

$$A=\begin{bmatrix} 2.5 & 8.1 & 0.4 & -9.9 \\ 5.1 & 0.0 & 2.3 & 2.5 \\ 2.1 & -1.0 & 1.2 & 3.4 \\ 3.4 & 2.1 & 1.0 & 2.3 \end{bmatrix} \rightarrow C=\begin{bmatrix} 2.3 & 3.4 & 2.5 & -9.9 \\ 1.0 & 1.2 & 2.3 & 0.4 \\ 2.1 & -1.0 & 0.0 & 8.1 \\ 3.4 & -11.1 & 5.1 & 2.5 \end{bmatrix}$$

Zatim, generisanom matricm C pomnožiti matricu B, pri čemu se dobija matrica D.

Rešenje zadatka br. 18

Zakonitost po kojoj se ova transformacija izvodi je:

T ($A_{I,J}$) –> $A_{N-J+1,N-i+1}$

```
Program Matrice;
Const       N=100;
            M=1000;
Paraconst   DIAG=N:(–1)1;
Var         A,C:array(1:N,1..N) of real;
            B:array(1:N,1..M);
            D:array(1:N,1..M) of real;
            I:Array(1:N) of integer;
            J,K:integer;
Index       P=1:N;

Begin
            With P do
    Begin
            I(#) := DIAG;
            For J:=0 to N–1 do
            C(#,I(# rotate –J) := A(# rotate –J, I(#));
            For K:=1 to M do
            D(#,K) := 0.0
            For J:=1 to N do
            For K:=1 to M do
            D(#,K) := D(#,K) + C(#,J)*B(J,K);
```

```
    End;
End.
```

Zadatak br. 19

U matrici $A_{M \times M}$ elementi (a_{IJ}) su jednaki jedinici ako za indekse tih elemenata važi: I=q*k+J, gde je Q konstanta i K=0,+−1,+−2,+−3.....dok su ostali elementi jednaki nuli.

Vektor C jednak je proizvodu matrice A i vektora B (AxB).

Napisati ACTUS program za generisanje vektora C na osnovu elemenata slučajno inicijalizovanog vektora B, pri čemu se ne koristi množenje i ispitivanje zavisnosti između indeksa elemenata matrice A.

Rešenje zadatka br.19

```
Program generisanje vektora;

Const       Q=10;

            M=100;

Index       P=1:(A)((M−1)divq * q+1);

Var         B,C:array (1:M) of integer;

            I,J:integer;

Begin

    For I:=1 to M do

            Begin

            Randomize;

            B(I:J) := random(1000);

            C(I:I) := 0;

            End;

    For J:=1 to (MdivQ) step Q do

            With P do

            For I:=0 to Q−1 do

            C(# shift I) := C(# shift I) + B(# shift I)

End.
```

Zadatak br. 20

Binarna Matrica A_{8kx8k} predstavlja šahovsku tablu od 64 polja, pri čemu je svako polje predstavljeno submatricom dimenzija KxK.

Svako belo polje predstavljeno je submatricom čiji svi elementi imaju vrednost 0, a svako crno polje submatricom čiji svi elementi imaju vrednost 1.

Polje na tabli u gornjem levom uglu je crno. Napisati program na ACTUS–u za generisanje matrice A.

Rešenje zadatka br. 20

```
Program sahovska_tabla

Const       K=10;

Var         A:array(1:8*K,1..8*K) of integer;

            I,J,L integer;

Index       S=1:(2K)6*K;

Begin

            For I:= 0 to 6 by 2 Do

            For J:=1 to K do

            For L:=0 to K–1 do

    Begin

            A(S shift L, I*K+J) :=1;

            A(S shift (I+k), I*K+J) :=0;

            A(S shift L,(I+1)*K+J) :=0;

            A(S shift (L+K),(I+1)*K+J) :=1;

    End;

End.
```

Zadatak br. 21

Napisati ACTUS program kojim se računa: F=DxB+ExC. Matrica D dobija se transponovanjem matrice A oko glavne dijagonale, a matrica E oko sporedne dijagonale. U dobijenoj matrici F dodeliti 0 svim elementima onih kolona koje sadrže bar jedan element koji ima vrednost M*N.

Rešenje zadatka br. 22

```
Program Matrice;

Const       N=100;

            M=200;

Parconst    Diag = 1:200;

            Spdiag=200:(–1)1;

Var         D,E,A:array )1:M,1..N) of integer;

            B,C,P,F:array(1:M) of integer;

            K:0..M–1

            I,J:integer;

Begin

    Within 1:N do

    Begin

    P(#) :=Diag;

    For k:=0 to M–1 do

    D(#,P(# rotate –K)) := A(# rotate –K, P(#));

End;

Randomize;

For I:=1 to N do

For J:=1 to M do

    Begin

            Randomize;

            A(I,J) := random (100);

    End;

For I:=1 to N do

    Begin

    Pom1(1:M,J) :=0;

    Pom2(1:M,J) := 0;

    For J:=1 to M do

            Begin

            Pom1(1:I,J) := pom1(1:I,J) + D(I,I:J)+B(I:J);
```

```
                Pom2(1:I,J) := pom2(1:I,J) + E(I:I,J)+C(I:J);
                End;
        End;
    For J:=1 to N do
        F(1:M,J) := 0;
                For I:=1 to M do
                F(1:I,J) := F(1:I,J) + pom1(1LI,J) + pom2(1:I,J)
                If ANY F(1:M,J) = M*N then
                F(1:M,J) := 0;
        End;
End.
```