Procesorski sistemi v telekomunikacijah

Jurij F. Tasič

Verzija 1.0, 28. maj 2004

Zahvala

Knjiga, ki je namenjena študentom tretjega letnika Fakultete za elektrotehniko, smer telekomunikacije, je odraz mojega dolgega dela z mikroprocesorji. Že na samem začetku mikroprocesorskih tehnologij, sem sledil razvoju štiri in kasneje osembitnih mikroprocesorjev. Pri tem sem se moral seznaniti z vsemi skrajnostmi mikroprocesorskih arhitektur, kar mi je omogočilo razvoj in izdelavo lastnega mikroprocesorja v diskretni obliki in kasneje tudi večih mikroračunalnikov.

Današnje tehnologije procesorjev daleč presegajo vsebino te knjige. Kljub temu mislim, da je za študente telekomunikacij potrebno dobro poznavanje mikroprocesorskih arhitektur, saj le-te najdemo v vsaki najmanjši aplikaciji na področju telekomunikacij, kot tudi na področju vse bolj popularnih intrektivnih multimedijskih sistemov.

Knjiga je nastajala kar nekaj let. Zapiske, ki sem jih uporabljal, sem dopolnil s prispevki kolegov, ki so ali študirali na katedri ali pa sem sodeloval pri njihovem raziskovalnem delu.

Dolga leta je bila knjiga nedokončana, nedokončan je bil vnos v računalnik, slike, pa tudi konceptualno. Pri slednjem so mi pomagali študentje višjih letnikov smeri telekomunikacije. Kljub vsem naporom, bi še knjiga ležala neobjavljena na moji delovni mizi, če ne bi srečal dveh mladih, nadobudnih študentov, ki sta mi pomagala pri dokončanju knjige. To sta Tomaž Finkšt in Andrej Kobal, katerima gre največja zahvala da je knjiga zagledala luč sveta.

Prav tako se moram zahvaliti mnogim neimenovanim kolegom, za sodelovanje pri raziskovalnem področju, katerega stranski rezultat je tudi ta knjiga.

Navsezadnje ne smem pozabiti na svojo soprogo dr. Darjo in moje tri sinove, Sašo, Gregorja in Andraža, ki so in še prenašajo moje muhe in težave, ki jih jim povzročam s svojim raziskovalnim in pedagoškim delom.

Jurij Franc Tasič

Predgovor

Knjiga Procesorski sistemi v telekomunikacijah...

Kazalo

Poglavje 1

Uvod

1.1 Komunikacijski sistemi

V svojem bistvu tako analogni kot digitalni komunikacijski sistemi služijo prenosu določenega niza informacij (podatkov, glasu ali slike), ki je v zveznih komunikacijskih sistemih predstavljen z zveznim signalom, v diskretnih komunikacijskih sistemih pa kot končni niz diskretnih sporočil. Glavna naloga sprejemnika v komunikacijskem sistemu ni samo v natančni reprodukciji izvirnega signala, pač pa v izločitvi oddanega signala iz sprejetega signala pomešanega s šumom, ki predstavlja npr. v primeru digitalnih komunikacijskih sistemov le eno obliko iz množice oddanih signalov. Zato je uporaba hitrih tehnik obdelave signalov vse pomembnejša predvsem v digitalnih komunikacijskih sistemih, kjer so pri prenosu signalov pomembni postopki oblikovanja in izvornega kodiranja, modulacije, ustreznega kodiranja za prenos signala po komunikacijskem mediju, multipleksiranja, hkratnega pristopa, enkripcije in sinhronizacije.

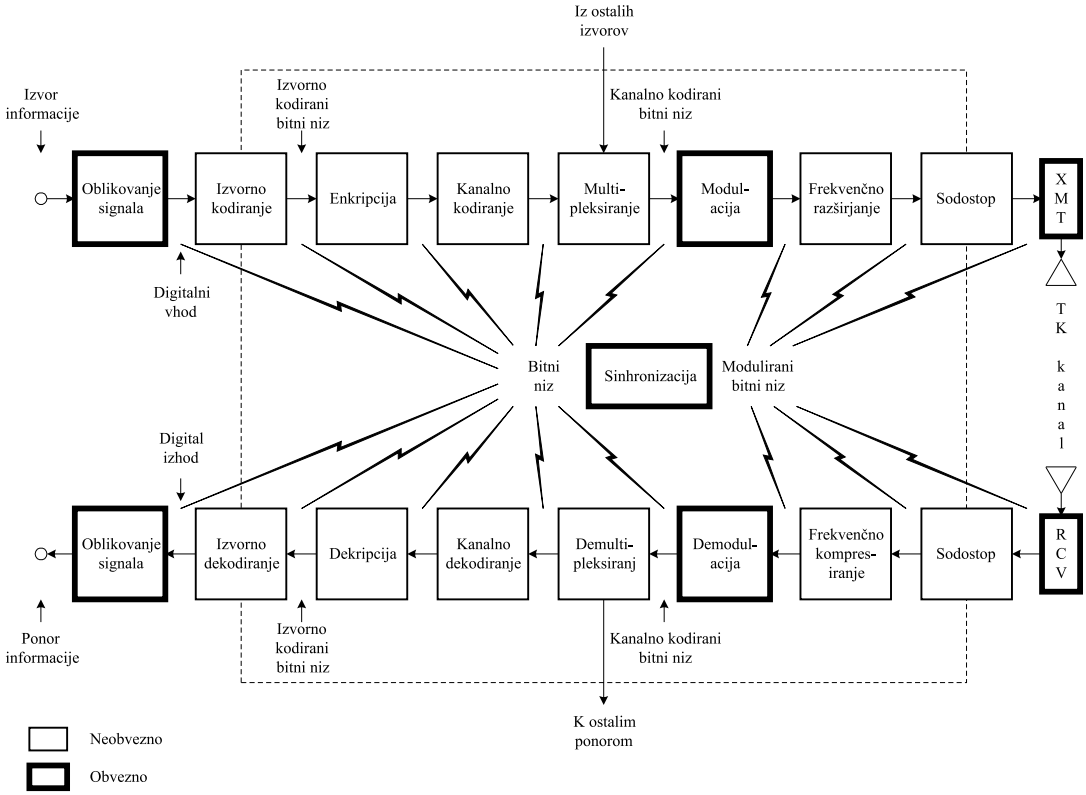
Digitalni komunikacijski sitemi so v svojem bistvu večprocesorski in večopravilni (večprocesni) sistemi, podprti z ustreznimi rešitvami, v katerih so uporabljeni standardni računalniški ali procesorski moduli ali specializirani signalni procesorji. Osnovna prednost digitalnih komunikacijskih sistemov pred analognimi sistemi je v lažji regeneraciji digitalnega signala v primerjavi z regeneracijo analognega signala. Digitalna vezja so bolj robustna in težje podvržena posameznim vplivom kot pa analogna vezja. Tako npr. v primeru binarnega digitalnega sistema poznamo le dve stanji tega sistema in sicer stanje 1 in stanje 0. V takem primeru mora biti zunanja motnja dovolj velika, da spremeni delovno stanje binarnega sistema iz stanja 1 v stanje 0 ali obratno. V nasprotju s tem pa analogni signali ne poznajo npr. dveh stanj, saj lahko zavzamejo poljubno mnogo možnih oblik. V tem primeru lahko celo najmanjše motnje povzroče nesprejemljiva popačenja analognega signala, ki ga na noben način ni moč popolnoma regenerirati.

V primerjavi z analognimi komunikacijskimi sistemi so digitalni sistemi zanesljivejši, hkrati pa je njihova izdelava cenejša kot je izdelava analognih komunikacijskih sistemov z enako zanesljivostjo prenosa informacij. Uporaba digitalnih vezij v komunikacijskih sistemih je zanesljivejša, stroški proizvodnje so nižji, uporaba programabilnih digitalnih enot, kot so mikroprocesorji ali signalni procesorji, pa omogoča večjo fleksibilnost komunikacijskih sistemov kot jo omogočajo analogni komunikacijski sistemi. Uporaba časovno porazdeljenega multipleksiranja (TDM - Time Division Multiplex)) pri kombiniranju prenosa različnih podatkov je cenejša, kot pa je prenos različnih analognih signalov z uporabo frekvenčno porazdeljenega prenosa (FDM - Frequency Division Multiplex). Najrazličnejši signali, ki predstavljajo podatke, govor ali slike, se v digitalnih prenosnih in komutacijskih sistemih obravnavajo enako, saj v vseh primerih bit predstavlja bit, zgolj zaradi enostavnosti pri prenosu pa se pogosto obravnavajo kot paketi. Digitalne tehnike teže k uporabi dodatnih funkcij obdelave signalov, ki ščitijo signal pred vplivi interference, omogočajo zaščito podatkov in preprečujejo nepooblaščen dostop z uporabo metod enkripcije. Ker velik del komunikacijskih uslug danes predstavlja prenos podatkov med dvema ali večimi računalniki ali digitalnimi merilnimi inštrumenti in računalnikom, je naravno,

da tudi v takšnih primerih uporabimo digitalne komunikacijske sisteme.

1.2 Obdelava signalov

Funkcionalni potek obdelave signala v tipičnem digitalnem komunikacijskem sistemu (DCS - Digital Communication System) je prikazan na sliki 1.1. Zgornji bloki prikazujejo elemente, ki se pojavljajo pri oddajnem delu prenosnega sistema, spodnji bloki pa predstavljajo elemente, ki jih potrebujemo za obdelavo signala v sprejemnem delu komunikacijskega sistema. Oba niza blokov sistema predstavljata hrbtenico in mišice komunikacijskega sistema. Vsi koraki, ki smo jih predvideli pri prenosu signalov so pomembni v sistemu prenosa signalov.



Slika 1.1: Blokovni diagram tipičnega digitalnega komunikacijskega sistema

Izvorno kodiranje predstavlja A/D pretvorbo analognega signala in odstranjevanje nepotrebnih (redundantnih) informacij. Enkripcija onemogoča nepooblaščen dostop do informacij, kanalno kodiranje pa zmanjšuje verjetnost napake PE ali kompleksnost dekoderja. Prav tako z uporabo tega postopka zožujemo potrebno frekvenčno širino prenosnega kanala. Multipleksiranje in sodostop kombinirata signale z različnimi karakteristikami ali celo signale iz različnih izvorov. Vsak korak obdelave signala v oddajni smeri se mora ponoviti v inverzni obliki v sprejemni smeri. Slično se tudi sprejeti signal pojavlja kot digitalni val vse dokler ga ne demoduliramo. Od tu naprej imamo opraviti z bitnim nizom primernim za nadaljno digitalno obdelavo. Na raznih mestih prenosa analognega signala se signalu doda tudi šum, kar povzroča, da moramo znati oceniti odposlani signal iz pokvarjenega sprejetega signala, ki ga dobimo na izhodu prenosnega sistema.

Iz tega sledi, da na komunikacijski poti obstaja več osnovnih postopkov digitalne obdelave signala, ki jih grupiramo v sledeče skupine:

- oblikovanje signala in izvorno kodiranje/dekodiranje (format, source encode/decode),

- enkripcija/dekripcija (encrypt/decrypt),
- kanalno kodiranje/dekodiranje (channel encode/decode),
- multipleksiranje/demultipleksiranje in sodostop (multiplex/demultiplex, multiple access),
- modulacija/demodulacija (modulate/demodulate),
- frekvenčno razširjanje/kompresiranje (frequency spread/despread),
- sinhronizacija (synchronization).

V posameznih fazah obdelave signala lahko uporabljamo posebna digitalna vezja, ali posebne hitre in najpogosteje tudi programabilne digitalne signalne procesorje (DSP - Digital Signal Processor), na določenih podsklopih pa celo hitre računalnike, primerne za uporabo v komunikacijskih sistemih.

Danes se v praksi pogosto pojavljajo tudi rešitve z uporabo posebnih namensko razvitih digitalnih procesorjev (ASIC - Application Specific Integrated Circuit) v VLSI tehnologiji, ki so prirejeni posameznim zahtevam uporabnika.

Poglavje 2

Enoprocesorski sistemi

2.1 Osnovni koncepti in von Neumannov računalniški model

Kot smo lahko razbrali iz uvodnega poglavja, diskretni sistemi vse pomembneje vplivajo na razvoj komunikacijskih sistemov, tako da se danes vse večji poudarek v praksi namenja ustrezni digitalni obdelavi signalov ali simulaciji ustreznih avtomatov, s katerimi lahko predstavimo kodirno vezje v prenosnem komunikacijskem sistemu ali telefonsko centralo v fiksnem komunikacijskem omrežju. Vsled vse večje uporabe programabilnih vezij in mikroprocesorjev v komunikacijskih sistemih si bomo v nadaljevanju ogledali osnovne koncepte delovanja mikroprocesorjev.

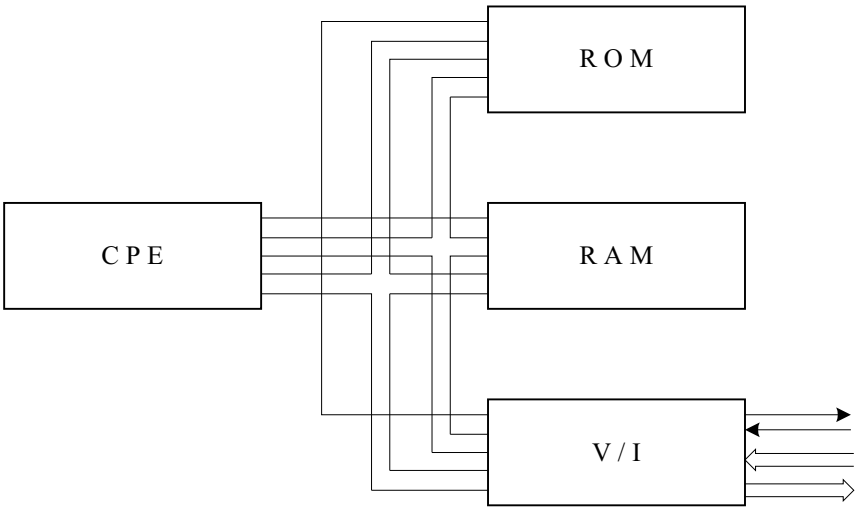
Ker je glavno vprašanje, ki se s tem v zvezi postavlja, kako mikroprocesor izvede določeno aritmetično, logično ali kakršnokoli drugo operacijo, če ga primerjamo s podobnimi diskretnimi rešitvami, si najprej oglejmo osnovno organizacijo mikroprocesorskega sistema. Vsak enoprocesorski von Neumannov računalnik ne glede na izjemo velike razlike pri tehnološki realizaciji in zmogljivostih sestavljajo:

- centralna procesna enota (CPE), ki obdeluje podatke na način, ki ji ga predpisuje niz ukazov, shranje v glavnem pomnilniku
- glavni pomnilnik, ki ga pogosto imenujemo tudi primarni ali hitri pomnilnik in je namenjen neposrednemu shranjevanju ukazov in operandov, ki jih pri svojem delu uporablja CPE ter je danes najpogosteje realiziran kot izključno bralni pomnilnik (ROM - Read Only Memory) ali pa kot bralno-pisanlni pomnilnik (RAM - Random Access Memory) in
- vhodno/izhodne (V/I) enote, ki so zadolžene za prenos podatkov v sistem ali iz njega.

Vsi osnovni gradniki von Neumannovega računalniškega modela so med seboj povezani preko podatkovnega, naslovnega in nadzornega vodila preko katerega le-ti med seboj komunicirajo.

Na tem mestu zgolj zaradi popolnosti opisa različnih računalniških modelov omenimo poleg von Neumannovega računalniškega modela, ki mu pogosto pravimo tudi ukazno-pretokovni model, saj dogajanje v takem modelu enolično določajo ukazi, ki jih CPE prevzema iz pomnilnika, podatkovno-pretokovni model, za katerega je značilno, da vrstni red operacij določajo operandi in ne ukazi. Ker pa podatkovno-pretokovni model računalniškega sistema v praksi še ni bil realiziran, se bomo v nadaljevanju omejili na obravnavo von Neumannovega modela mikroračunalnika, ki je prikazan na sliki 2.1.

Programe, ki se izvajajo v mikroračunalniku, sestavljata dve povsem različni vrsti podatkov: prvi vrsti podatkov pravimo ukazi in določajo kakšne vrste operacijo naj CPE izvrši nad podatki druge vrste, ki jim pravimo operandi in so najpogosteje kar podatki sami, takšnemu načinu podajanja operandov pravimo takojšnje naslavljanje (immediate addressing), ali pa njihovi naslovi v pomnilniku, takšen način podajanja operandov pa imenujemo neposredno naslavljanje (direct addressing). Glede na število eksplicitnih operandov v najpomembnejših ukazih, predvsem tistih, ki določajo delovanje ALE, torej ločimo m-operandne mikroračunalnike.



Slika 2.1: Blokovna shema von Neumannovega računalniškega modela

Če torej povzamemo, je za von Neumannov računalniški model, ki ga prikazuje slika 2.1, kateremu pripadajo danes praktično vsi delujoči računalniški sistemi, značilno da ga sestavljajo tri osnovne enote: CPE, glavni pomnilnik in V/I enote in da njegovo delovanje določa v glavnem pomnilniku shranjeni program, katerega ukazi se izvajajo zaporedoma drug za drugim.

2.2 Centralna procesna enota

Čeprav se posamezne CPE od proizvajalca do proizvajalca močno razlikujejo, imajo vse skupne nekatere osnovne lastnosti, ki jih bomo v nadaljevanju predstavili na modelu 8-bitnega navideznega procesorja, ki je vsaj v principu delovanja zelo podoben enemu prvih mikroprocesorjev Motoroli MC6800.

2.2.1 Registri CPE

Akumulator

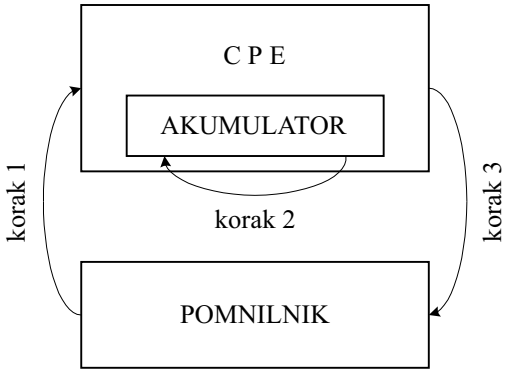
Akumulator je register, ki se nahaja v vsaki CPE in je kot eden od programsko dostopnih registrov najpogosteje namenjen shranjevanju rezultatov aritmetičnih ali logičnih operacij ter prenosu podatkov iz pomnilnika v CPE. V večnivojski pomnilniški hierarhiji, ki jo danes v praksi uporabljajo vsi mikroračunalniški sistemi, je predvsem zaradi tehnološke izvedbe v jedru mikroprocesorja akumulator glede na dostopni čas najvišje. V nadaljevanju predpostavimo, da ima naš navidezni procesor en sam 8-bitni akumulator, pa čeprav današnji moderni procesorji vsebujejo tudi že 64-bitne akumulatorje.

Akumulator

7	6	5	4	3	2	1	0

Navidezna CPE deluje direktno nad vsebino akumulatorja in ne izvaja aritmetičnih ali logičnih operacij nad podatki v pomnilniku. Seveda takšen pristop zahteva, da navidezna CPE izvede vsaj tri korake pri izvajanju enega ukaza, kar je prikazano na sliki 2.2.

Tovrstni enooperandni ukazi pa v praksi niso zadostni. Ker nekateri ukazi navidezne CPE potrebujejo vsebini dveh pomnilniških besed, ki jih morajo ločeno prebrati iz pomnilnika, pomeni to v praksi pogosto veliko izgubo dragocenega procesorskega časa, tako da imajo v ta namen praktične CPE pogosto vgrajene še dodatne akumulatorske registre, ki služijo vmesnemu shranjevanju podatkov.



Slika 2.2: Osnovni koraki pri izvajanju ukaznega niza

Podatkovni števec

Podatkovni števec omogoča dostop CPE do poljubne pomnilniške lokacije, saj z njim naslavljamo lokacijo v pomnilniku iz katere namerava prebrati ali vpisati podatek.

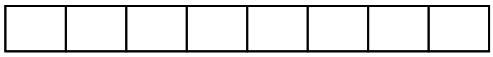
Podatkovni števec

Nekatere CPE imajo tudi več podatkovnih števcev. Zaradi enostavnost predpostavimo, da imamo le en 16-bitni podatkovni števec, s katerim naslavljamo $2^{16} = 65536$ pomnilniških lokacij. Podobno, kot potrebujemo akumulator za shranjevanje podatkov, potrebujemo tudi register, v katerem bomo shranjevali kode ukazov, prebranih iz pomnilniške lokacije.

Ukazni register

Ukazni register omogoča shranjevanje posameznih ukazov. CPE njegovo vsebino obravnava kot ukaz (instruction).

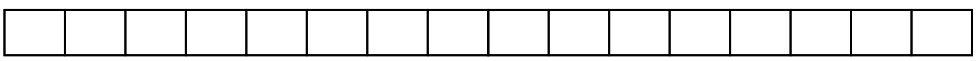
Ukazni register



Programski števec

Programski števec določa naslov pomnilniške lokacije s katere CPE prebere kodo ukaza. Programski števec je analogen podatkovnemu števcu, le da je ta vedno namenjen branju podatkov iz naslovljene pomnilniške podatkovne lokacije, prvi pa je vedno namenjen branju kode ukaza iz naslovljene pomnilniške lokacije kode ukaza.

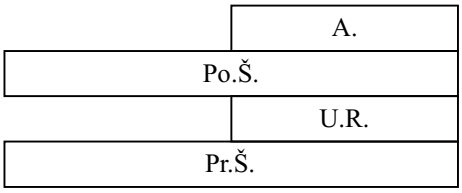
Programski števec



Med podatkovnim in programskim števcem obstoja konceptualna razlika. Pri vpisovanju kod ukazov v zaporedne pomnilniške lokacije pri uporabi programskega števca ni nikakršnih težav, ki bi lahko

nastopile v obratnem primeru. Prav tako je pomembno, da vedno ob *resetiranju* ali ponovnem zagonu sistema postavimo v ta programski števec vedno isto vsebino - *začetni naslov*. Po vsakem branju kode ukaza iz pomnilnika se vsebina programskega števca razen pri skočnih ukazih poveča za 1. Tako programski števec vedno kaže na pomnilniško lokacijo naslednje kode ukaza.

V primeru, ko pa imamo opraviti z zaporedjem ukazov, tedaj s podatkovnim števcem zaporedno dostopamo do podatkov v pomnilniku. Tudi če posegamo le po enem samem podatku nikoli ne vemo vnaprej kakšna naj bo vsebina tega podatkovnega števca.



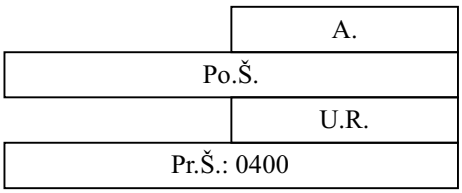
Slika 2.3: Registri enostavne CPE

Uporaba registrov CPE

Zaradi lažjega razumevanja uporabe registrov CPE si oglejmo enostaven binarni program seštevanja dveh besed z uporabo navedenih registrov. Vsebina programa je naložena npr. v pomnilniku na lokaciji 0400_{16}. V začetku nastavimo vsebino programskega števca Pr.Š. na 0400_{16}, ki kaže na prvi ukaz programa shranjenega v pomnilniku. Vzemimo, da se podatka $7A_{16}$ in $2F_{16}$ nahajata na pomnilniških lokacijah $0A30_{16}$ in $0A31_{16}$.

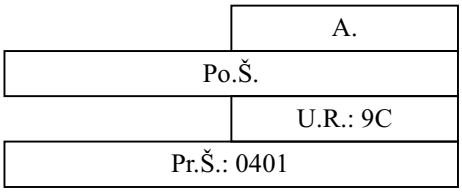
naslov v pomnilniku	vsebina lokacije	
0400	9C	1. ukaz
0401	0A	
0402	30	
0403	40	2.ukaz
0404	9C	3.ukaz
0405	0A	
0406	31	
0407	80	4.ukaz
0408	60	5.ukaz
0409		
	⋮	
0A30	7A	
0A31	2F	

Vsebine posameznih registrov:



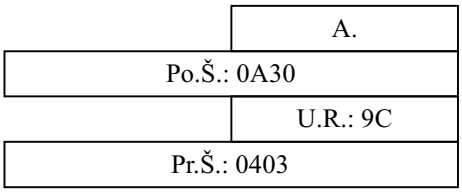
CPE enota sedaj naloži vsebino lokacije na katero kaže vsebina programskega števca Pr.Š. v ukazni register U.R. Takoj zatem CPE poveča vsebino programskega števca.

Vsebine posameznih registrov:



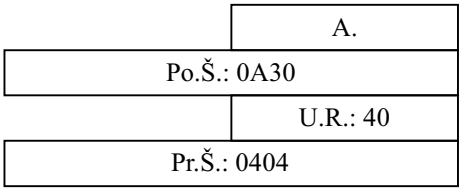
Koda 9C, ki se je prenesla v ukazni register povzroči, da logika CPE enote izvede dva koraka. V prvem se vsebina pomnilniške lokacije, na katero kaže programski števec, prenese v zgornjo besedo podatkovnega števca. Takoj za tem se poveča vsebina programskega števca, nakar se izvede naslednji korak ukaza, ki prenese vsebino naslednje pomnilniške lokacije v spodnjo besedo podatkovnega števca. Takoj za tem prenosom se vsebina programskega števca zopet poveča.

Vsebine posameznih registrov:



Tako se je izvršil prvi ukaz. Nadaljujmo z drugim ukazom. Po izvršitvi prvega ukaza CPE prebere vsebino spominske lokacije na katero kaže programski števec. Vsebino, ki jo procesor obravnava kot ukaz, shrani v ukazni register.

Vsebine posameznih registrov:



Ta ukaz povzroči branje vsebine lokacije, na katero kaže podatkovni register v akumulator A.

Vsebine posameznih registrov:

	A.: 7A
Po.Š.: 0A30	
	U.R.: 40
Pr.Š.: 0404	

Po izvršitvi drugega ukaza, se poveča vsebina programskega števca, ki kaže na kodo naslednjega ukaza. Tretji ukaz predstavlja ponovitev prvega ukaza le, da je vsebina podatkovnega registra $0A31_{16}$.

Vsebine posameznih registrov:

	A.: 7A
Po.Š.: 0A31	
	U.R.: 9C
Pr.Š.: 0407	

Po zaključku tretjega ukaza se prične izvajanje četrtega ukaza. Kot do sedaj CPE avtomatsko prenese vsebino iz naslovljene pomnilniške lokacije v ukazni register.

Vsebine posameznih registrov:

	A.: A9
Po.Š.: 0A31	
	U.R.: 80
Pr.Š.: 0408	

Koda ukaza 80_{16} zahteva branje vsebine pomnilniške lokacije naslovljene s podatkovnim števcem in prištetje te vsebine k vsebini akumulatorja A.

Rezultat seštevanja želimo shraniti v pomnilniški register naslovljen s podatkovnim števcem, to je na lokacijo $0A31_{16}$. To operacijo izvede CPE s petim ukazom 60_{16}. Vsebina akumulatorja se na omenjeno lokacijo prenese v dveh korakih:

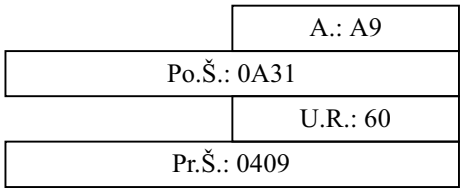
- v prvem koraku prebere CPE ukaz iz pomnilnika

V tem primeru so vsebine posameznih registrov:

	A.: A9
Po.Š.: 0A31	
	U.R.: 60
Pr.Š.: 0409	

- v drugem koraku pa se shrani vsebina akumulatorja na pomnilniško lokacijo naslovljeno s podatkovnim registrom.

Vsebine posameznih registrov so:



2.2.2 Aritmetično-logična enota

Obdelava podatkov se dejansko vrši v skupini logičnih komponent in v aritmetično-logični enoti ALE mikroprocesorja. ALE izvaja operacije nad 8-bitnimi binarnimi podatki, prav tako pa tudi potrebuje ustrezno podporo logičnih vezij za izvajanje operacij kot so:

- binarno seštevanje,
- Boolove operacije,
- računanje komplementa podatkovne besede,
- pomikanje besed za en bit levo ali desno.

2.2.3 Nadzorna enota

Nadzorna enota NE mikroprocesorja razporeja delo posameznih enot ALE tako, da se ukaz pravilno izvede. Nadzorna enota deluje tako kot ji to določa vsebina ukaznega registra. To pomeni, da se vsebina ukaznega registra vsakokrat dekodira v NE, ki glede na bitni vzorec kode ukaza generira zaporedja izbirnih (enable) signalov, kateri omogočijo pravilni pretok podatkov skozi ALE ali pa v danih trenutkih omogoči delovanje posebnih ALE logičnih enot. Mikroprocesor, ki bi zadostoval vsem omenjenim zahtevam je prikazan na sliki 2.4.

Register stanj CPE

CPE mora poleg že omenjenih registrov vsebovati register, ki v obliki posameznih bitov shranjuje rezultate logičnih in aritmetičnih operacij. Vsak bit v tem registru imenujmo *zastavica stanja*.

- Zastavica prenosa (carry flag) je odraz večbesednih aritmetičnih operacij in vsak prenos bita z največjo utežjo utežjo zaznamo s stanjem 1, na mestu bita zastavice prenosa v registru stanj.

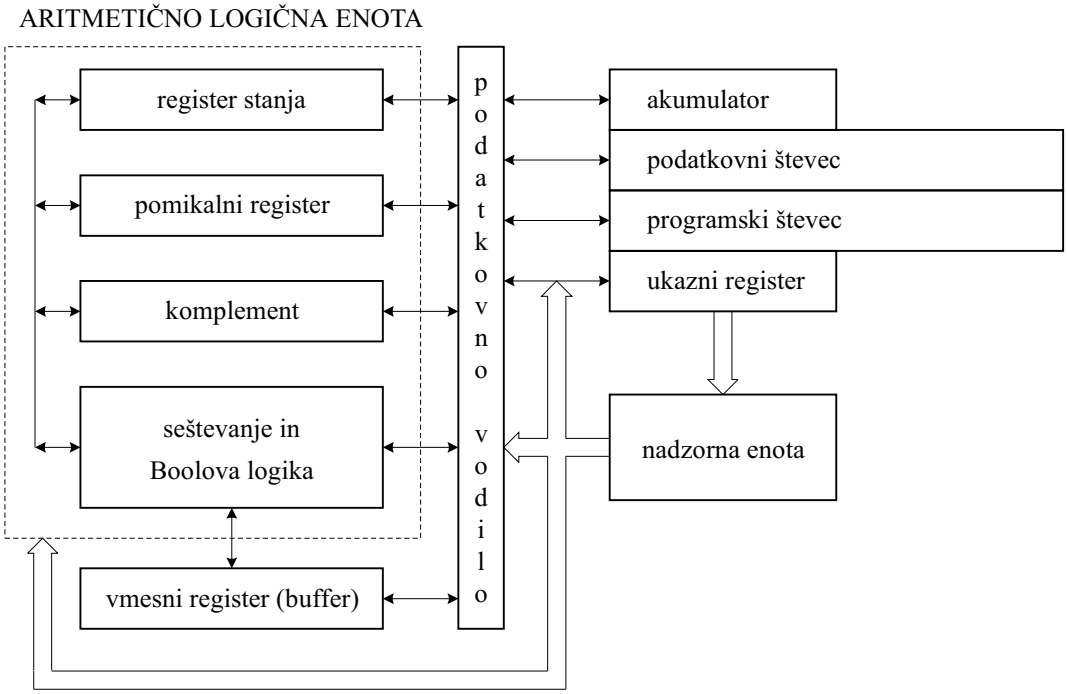
  V primeru izvajanja BCD aritmetike v CPE, imamo opravka z zastavico prenosa med štirimi biti (nibble).

- Ničelna zastavica (zero flag) ponazarja v primeru, da je v stanju 1, da je bil rezultat operacije nad podatki nič. Ukaze, s katerimi postavljamo ali brišemo zastavice stanj, moramo pazljivo izbrati

- Zastavica predznaka operacije (sign flag), določa ali je rezultat operacije nad podatki negativen. Pri vsakem eno ali večbesednem podatku je znaka vedno bit z največjo utežjo.

- Stanje prekoračitve (overflow flag) označuje dogodek, ko je rezultat operacije nad podatki prevelik, da bi ga lahko shranili v določenem registru z omejenim številom bitov.

- Stanje paritete (parity flag) je vedno 1, ko operacija prenosa zazna podatek z napačno pariteto.

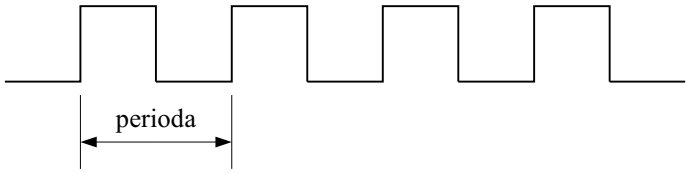


Slika 2.4: Funkcijska soodvisnost enot CPE in nadzorne enote

Izvajanje ukazov

Kot smo opisali, CPE lahko interpretira informacijo shranjeno v pomnilniku kot podatek ali kodo ukaza. Pri tem pa si nismo odgovorili kaj se dogaja s CPE enoto v fazi izvajanja neke operacije. Če si želimo odgovoriti na to vprašanje moramo vpeljati osnovni koncept mikroračunalnika, s podporo ustrezne zunanje logike, ki je lahko vgrajena v neko VLSI vezje, ali pa jo po potrebi določi uporabnik. Oglejmo si najprej kakšno zunanjo podporo podtrebuje mikroporocesor za izvajanje ukazov.

Sistemska ura je potrebna za nadzorovano izvajanje posameznih operacij. Za označitev urinega signala, ki je prikazan na sliki 2.5, uporabljamo simbol Φ.



Slika 2.5: Urin signal

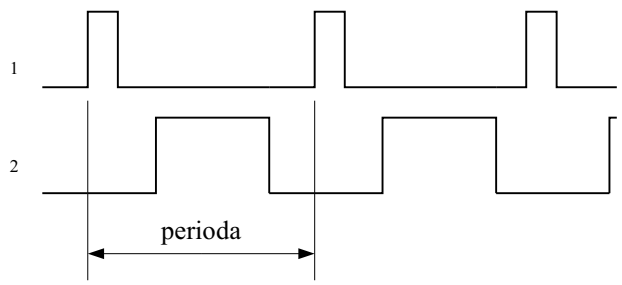
Glede na arhitekturo CPE je lahko signal ure, ki poganja delovanje enojen, v nekaterih primerih pa se uporablja kompleksnejša povezava urinih signalov. Tak primer soodvisnosti dveh urinih signalov Φ_1, Φ_2 je prikazan na sliki 2.6.

V slednejm primeru imamo tri stanja na osnovni cikel dveh urinih signalov in sicer:

$$\begin{array}{lcccc} \Phi_1 & = & 1 & 0 & 0 \\ \Phi_2 & = & 0 & 0 & 1 \end{array}$$

Ukazni cikli

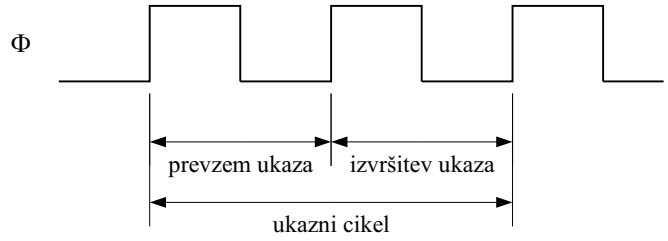
Mikroračunalniško izvajanje vsakega ukaza lahko razdelimo na dva dela:



Slika 2.6: Dva urina signala

- branje ukaza in operandov iz pomnilnika,
- izvajanje operacijske kode ukaza.

V fazi branja ukaza mora logika CPE poleg naslavljanja pomnilnika z vsebino programskega števca, še zagotoviti generiranje ustreznih krmilnih signalov, ki določajo, da mora zunanja logika vrniti CPE vsebino omenjene pomnilniške lokacije. Tej operaciji pravimo *fetch* - operacija branja. Ta vsebina se shrani v ukazni register CPE. CPE uporablja svojo notranjo logiko za povečevanje vsebine programskega števca za 1. Za izvedbo ukaznega cikla torej potrebujemo dva urina cikla:



Slika 2.7: Ukazni cikel

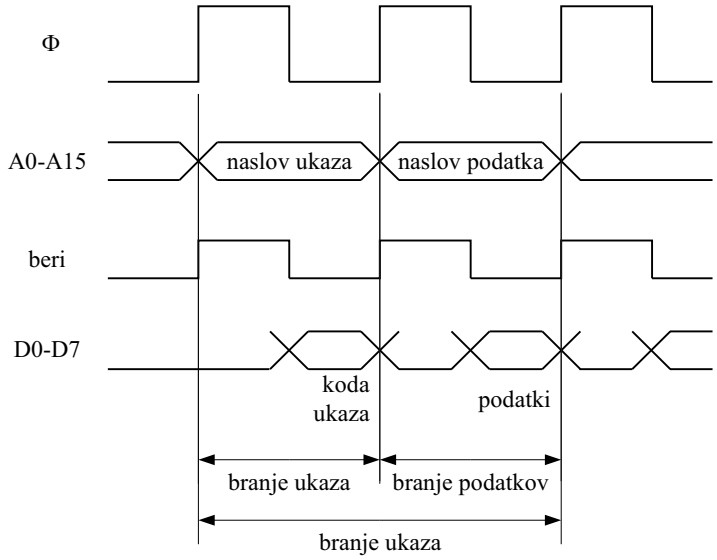
Mikroprocesor komunicira preko ustreznih priključkov s pomnilniškimi in vhodno/izhodnimi enotami. Enote naslavlja preko 16-bitnega vodila, podatke pa bere preko dvosmernega 8-bitnega vodila z uporabo dodatnega beri/piši (*read/write*) krmilnega signala. Ustrezen potek signalov ukaza *memory read* je razviden s slike 2.8.

Edina razlika med ukazoma za branje iz pomnilnika in za vpis podatkov v pomnilnik je le v krmilnem signalu beri (*read*) oziroma piši (*write*).

Ukaz *vstavi podatkovni števec* - (*Load Data Counter*) dejansko vpiše dve osem bitni besedi, ki slede sami kodi ukaza. Ta ukaz lahko razdelimo na dva podukaza, kjer prvi vpiše bolj pomembno besedo v podatkovni števec in drugi manj pomembno besedo v isti podatkovni števec. V osnovnem primeru imamo opraviti le z enim ukazom in dvema besedama z naslovom, v drugem primeru pa z dvema ukazoma in z dvema podatkoma, ki pripadata po enemu izmed ukazov.

Mikroprogrami in nadzorna enota

Oglejmo si kako nadzorna enota dekodira operacijsko kodo ukaza. Mikroprocesor s slike 2.4 moramo za normalno delovanje računalnika dopolniti še s še s kontrolnimi elementi, kot tudi z drugimi elementi Izvajanje vsake mikrokode sproži poseben signal iz nadzorne enote. Zapovrstno nizanje mikroukazov vodi k makroukazu, ki je nekakšen odziv CPE na ukaz zbirnika. Torej je mikroprogram le niz binarnih kod shranjenih znotraj nadzorne enote CPE. Iz tega vidimo, da obstoja tesna povezava med mikroprogramiranjem in programiranjem v zbirnem jeziku. Vsaka koda ukaza makroprograma (ukaza zbirnika) namreč sproži izvajanje pripadajočega mikroprograma shranjenega v nadzorni enoti. Posamezni ukazi



Slika 2.8: Prikaz poteka signalov pri branju podatkov iz pomnilnika

mikroprograma pa sprožijo izvajanje posameznih logičnih operacij znotraj same CPE. Dejansko izvajanje se potemtakem v vsakem mikroračunalniku vrši na nivoju mikrokode, oz. mikroprograma. Če smo kot uporabniki sposobni spremeniti mikrokodo znotraj nadzorne enote, pravimo, da je mikroprocesor mikroprogramirljiv. V primeru, ko pa je mikroprogram nespremenljiv, imamo opraviti s standardnim nemikroprogramabilnim mikroprocesorjem.

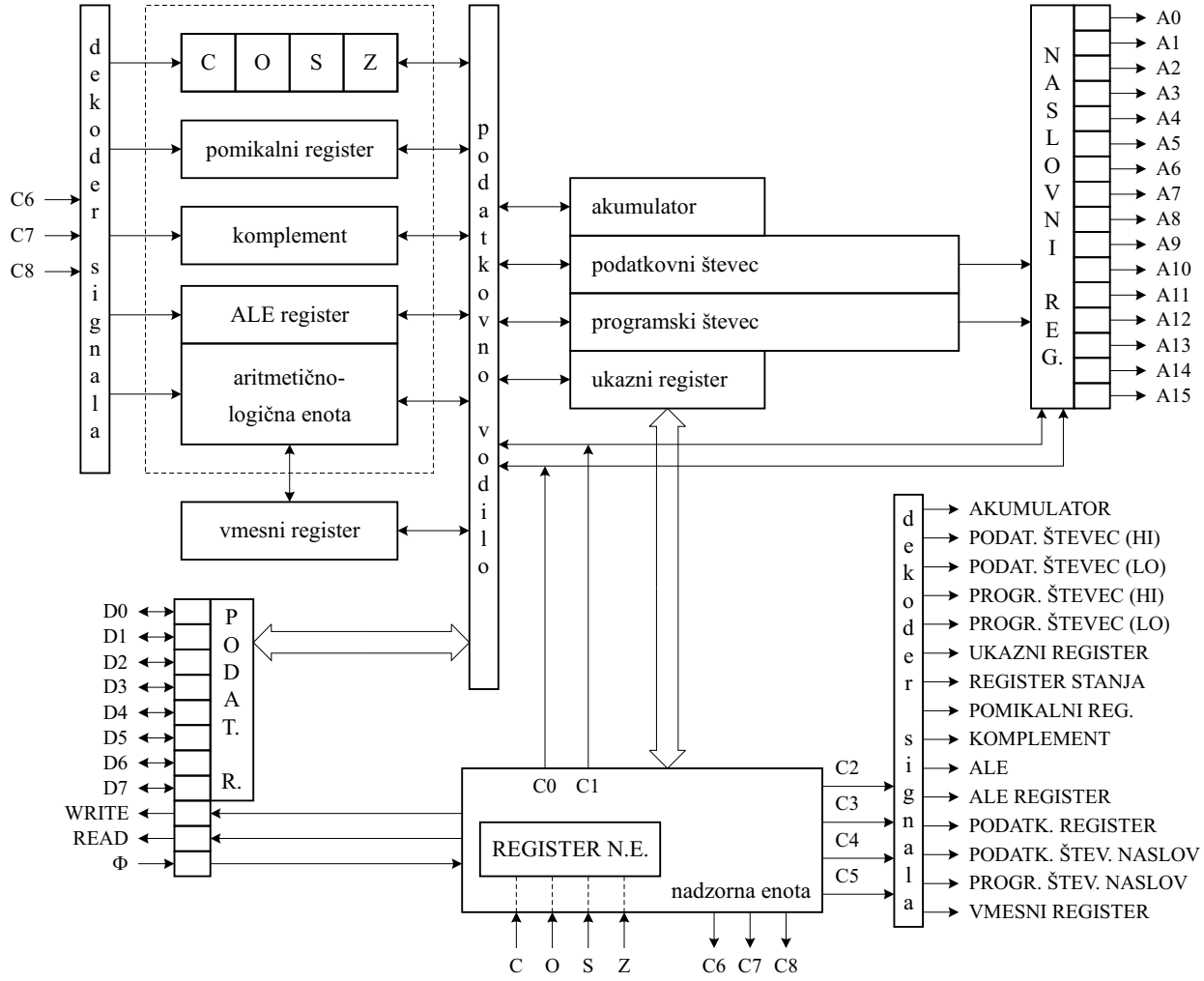
Zato dopolnimo sliko 2.4 z ustreznimi povezavami, signali in nadzorno enoto.

S slike 2.9 je razvidno, da smo v miroprocesor vpeljali več kontrolnih signalov. Nadzorna enota naše CPE uporablja te krmilne signale pri izvajanju posameznih makroukazov. Oglejmo si v tabelah podane naloge posameznih krmilnih signalov.

KRMILNI SIGNAL	NALOGE
C0, C1	C0=0, C1= 0 ni prenosa podatkov v ali iz podat. vodila ali naslovna vodila C0=1, C1= 0 prenos podatkov na podatkovno vodilo ali v naslovna vodila C0=0, C1= 1 prenos podatkov iz podatkovnega vodila ali naslovna vodila C0=1, C1= 1 mikrokoda nadzorne enote (tabela 2.4)
C2, C3, C4, C5, C6, C7, C8	ko je C0=1, C1= 0, ali C0=0, C1= 1, se sproži prenos podatkov določen v tabeli 2.2 Signali C6, C7, C8 so uporabljeni v ALE za dekodiranje funkcij določenih v tabeli 2.3
Φ	urin signal na vhodu nadzorne enote
C,O,S,Z	štirje biti stanj, povezani na podatkovno vodilo nadzorne enote

Tabela 2.1: Signali nadzorne enote

Krmilni signali v tabelah 2.1, 2.2, 2.3 niso zadostni, da bi lahko nadzorna enota izvajala vse ukaze zbirnika, saj v tabelah ni opaziti ukazov branja in pisanja v pomnilnik ter aktivnosti z zastavicami C, O, S, Z.



Slika 2.9: Signali nadzorne enote navideznega mikroračunalnika

V primeru, če sta C0 in C1 oba v stanju 1, tedaj prevzamejo ostali signali od C2 do C8 določene krmilne funkcije pri delovanju nadzorne enote CPE. Te razvrstimo v pet skupin.

- V prvi skupini so vrednosti krmilnih linij od C2 do C8 enake 0. V tem primeru se register stanj in s tem biti S, Z, O in C, prenese v podatkovni register kontrolne enote.

- V primeru ko sta C2 = 1 in C3 = 0, potem C5, C6, C7, C8 vsebujejo pripadajoče vrednosti bitov registra stanja in sicer Z, S, O, C.

- Če sta C2 = 0 in C3 = 1, tedaj se testirajo posamezni biti registra stanj na vrednost 0 in ta pogoj omogoča preskok naslednejga ukaza.

- Če so vrednosti bitov C2 = 1, C3 = 1, C4 = 1, tedaj določajo C5, C6, C7, C8 stanje poljubnih nadzornih signalov, ki jih vodimo na priključke VLSI vezaj CPE. Signale *Read, Write* pa lahko generiramo na C8 ali C7 krmilni liniji.

- V primeru, ko pa so C2, C3, C4 v stanju 1, tedaj se C5 do C8 interno dekodirajo (v nadzorni enoti) in določajo eno od šestnajstih logičnih operacij znatraj CPE (tabela 2.2).

C2	C3	C4	C5	NALOGE
0	0	0	0	Akumulator ↔ Podatk. vodilo izbrano
0	0	0	1	Podatk. števec - HI beseda ↔ Podatk. vodilo izbrano
0	0	1	0	Podatk. števec - LO beseda ↔ Podatk. vodilo izbrano
0	0	1	1	Progr. števec - HI beseda ↔ Podatk. vodilo izbrano
0	1	0	0	Progr. števec - LO beseda ↔ Podatk. vodilo izbrano
0	1	0	1	Ukazni register ↔ Podatkovno vodilo izbrano
0	1	1	0	Register stanj ↔ Podatkovno vodilo izbrano
0	1	1	1	Pomikalni register ↔ Podatkovno vodilo izbrano
1	0	0	0	Komplement. r. ↔ Podatkovno vodilo izbrano
1	0	0	1	ALE zatiči ↔ Podatkovno vodilo izbrano
1	0	1	0	ALE register ↔ Podatkovno vodilo izbrano
1	0	1	1	Podatkovni register ↔ Podatkovno vodilo izbrano
1	1	0	0	Podatkovni števec ↔ Naslovno vodilo izbrano
1	1	0	1	Programski števec ↔ Naslovno vodilo izbrano
1	1	1	0	Podatkovni register ↔ Vmesni register
1	1	1	1	ni uporabljeno

Tabela 2.2: Pretok podatkov, ko sta C0 = 1 ali C1 = 1

C6	C7	C8	NALOGE
0	0	1	Logika pomikalnega registra izbrana
0	0	0	Logika komplem. registra izbrana
0	1	1	Logika za seštevanje izbrana
0	1	0	AND logika izbrana
1	0	1	OR logika izbrana
1	0	0	XOR logika izbrana
1	1	1	Povečaj vsebino zatičev (latches) ALE
1	1	0	NOP ALE

Tabela 2.3: Izbirni signali ALE

Mikroukazi

Oglejmo si enostaven mikroprogram branja podatka iz pomnilnika. V našem primeru enostavnega modela CPE vsak mikroukaz, ki je vgrajen v nadzorno enoto CPE, sestavlja devet bitov. Mikroprogram nadzorne enote nikoli ne vsebuje ukazov, zato je dolžina *besede* mikrokode lahko poljubna. V tabeli 2.2 je tabeliranih petnajst miroukazov *fetch* makroukaza, ki jih lahko shranimo v matriki dimenzije 15 × 9. Zavedati se moramo, da se vsak mikroukaz izvrši znotraj fetch ukaza, torej v eni sami periodi sistemske ure. To pomeni, da nadzorna enota razdeli vsak interval na 16 podintervalov.

Eden enostavnejših, a učinkovitih procesorjev, ki se je močno uveljavil v letih 1974-1980 poleg bralnega procesorja INTEL 8080 je bil tudi procesor MOTOROLA 6800. Malo kasneje se je pojavil še procesor Z80, z naborom pozitivnih lastnosti njegovih predhodnikov. Vsi primeri s področja enoprpcesorskih sistemov, predstavljenih v tem učbeniku, se bodo nanašali na enega od omenjenih procesorjev. Motorolina CPE je predtavljala ugodno razmerje med registri, močjo ukazov zbirnika in kontrolno logiko. Na sliki 2.10 so predstavljeni osnovni registri CPE MOTOROLA 6800.

Št. ukaza	Koda mikroukaza									NALOGA
	C8	C7	C6	C5	C4	C3	C2	C1	C0	
1	1	1	1	1	0	1	1	0	1	premakni vsebino prog. števca v naslov. register
2	1	0	0	0	0	1	1	1	1	postavi READ signal v 1, WRITE v 0
3	1	1	1	0	0	1	0	0	1	premakni LO vseb. prog. števca na podat. vodilo
4	1	1	1	1	0	0	1	1	0	premakni pod. vodilo na zatič (latch) ALE
5	0	1	1	0	0	0	0	0	0	povečaj vsebino zatičev ALE
6	1	1	1	1	0	0	1	0	1	premakni vsebini ALE zatičev na podat. vodilo
7	1	1	1	0	0	1	0	1	0	premakni podat. vodilo v LO program. števca
8	1	1	1	1	1	0	0	0	1	premakni HO progr. števca na podatkovno vodilo
9	1	1	1	1	0	0	1	1	0	premakni podat. vodilo v zatiče (latches) ALE
10	1	0	0	0	0	1	0	1	1	preskoči nasl. mikrokodo, če je Carry bit = 0
11	0	1	1	0	0	0	0	0	0	povečaj vsebino zatičev ALE
12	1	1	1	1	0	0	1	0	1	premakni vseb. zatičev ALE na podat. vodilo
13	1	1	1	1	1	0	0	0	1	premakni podat. vodilo v HO progr. števca
14	1	1	1	1	1	0	1	0	1	premakni podat. reg. na podatkovno vodilo
15	1	1	1	1	0	1	0	1	0	premakni podatkovno vodilo na ukazni register

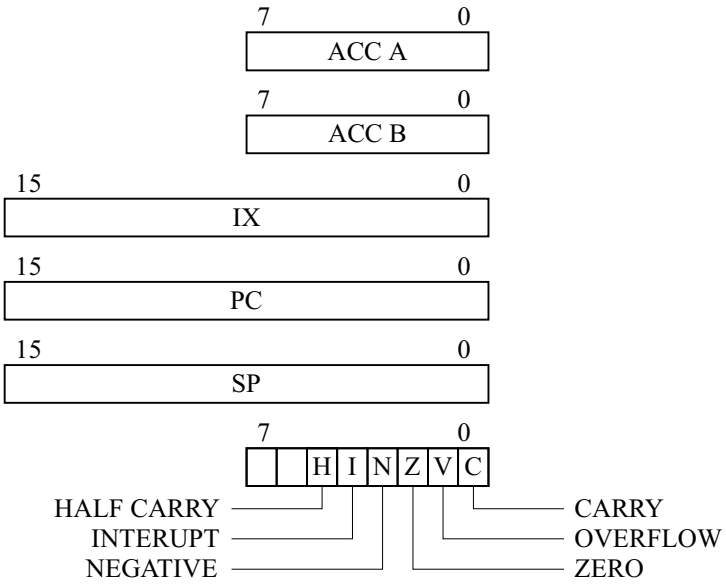
Tabela 2.4: Mikroprogram zbirnega ukaza beri iz pomnilnika

2.3 Zunanja logika, elementi in navidezni procesor

Kot smo že uvodoma ugotovili moramo zunanje elemente (RAM,ROM, V/I enote) vedno ločeno izbirati, za kar potrebujemo določene logične podsklope.

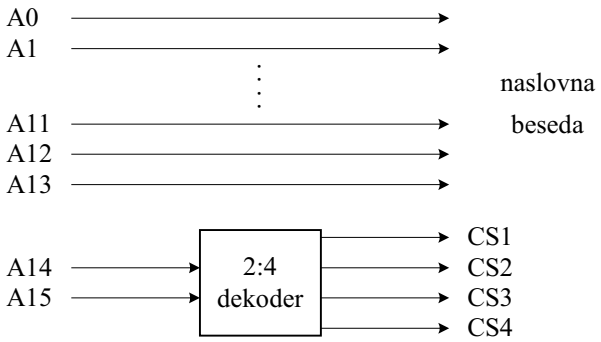
Zunanji pomnilnik je najpomembnejši element za pravilno delovanje CPE, zato si najprej poglejmo povezavo enostavnega ROM Read Only Memory elementa s CPE. Signali, ki jih zato potrebujemo so zelo enostavni:

- naslovno vodilo za poseganje do vseh lokacij,
- krmilni signal *read* - (beri), ki določa, kdaj naj se pojavi vsebina pomnilniške lokacije na vodilu,
- sinhronizacijski signal Φ,
- napajanje.



Slika 2.10: Programski model procesorja MOTOROLA 6800

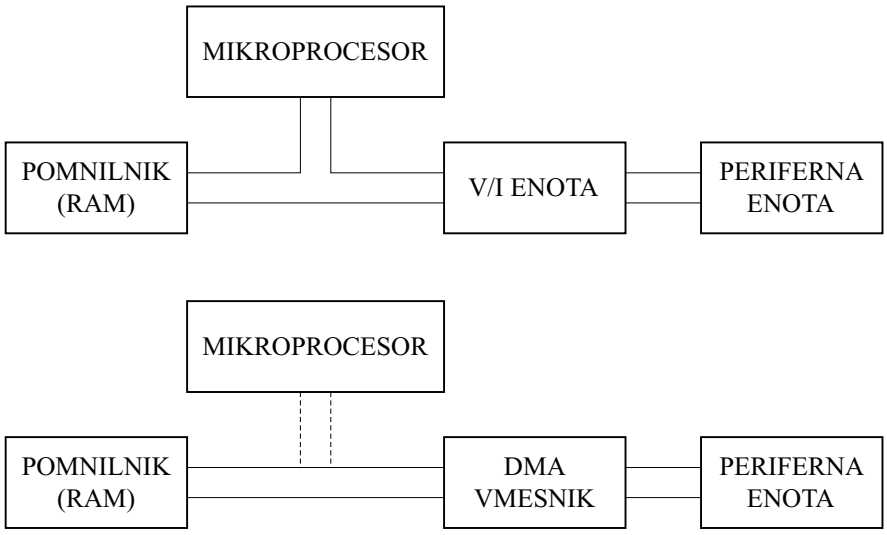
Vsak ROM element ima logiko za izbor pomnilniškega elementa (CS), ter ustrezno število naslovnih linij. Običajno se linija iz dekodirnega elementa, ki omogoča izbor pomnilniškega elementa, postavi v logično stanje 0, dočim ostali izhodi ostanejo v stanju 1. Če vzamemo pomnilnik s 16 K besedami pomnilniških celic, tedaj potrebujemo naslovne linije od A0 do A13, dočim se naslovni liniji A14 in A15 uporabita pri logiki za izbor pomnilniškega elementa (slika 2.11).



Slika 2.11: Naslavljanje in dekodiranje ROM elementa

Podoben postopek velja tudi za izbiro pomnilnika RAM (Random Access Memory), v katerega lahko vpisujemo podatke ali pa jih iz njega tudi beremo, prikazuje slika 2.8. Logika za izbiro pomnilnika mora biti sposobna izločiti podatke iz naslovljene pomnilniške lokacije, in te ob pravem času postaviti na zunanje podatkovno vodilo. Poleg zunanjih podatkovnih linij in zunanjih linij moramo upoštevati še signalni liniji za vpis in izpis *Write/Read*, napajalni liniji in linijo za sinhronizacijo Φ. Kot v predhodnem primeru izberemo niz pomnilniških elementov z izbirno linijo CS (*chip select*).

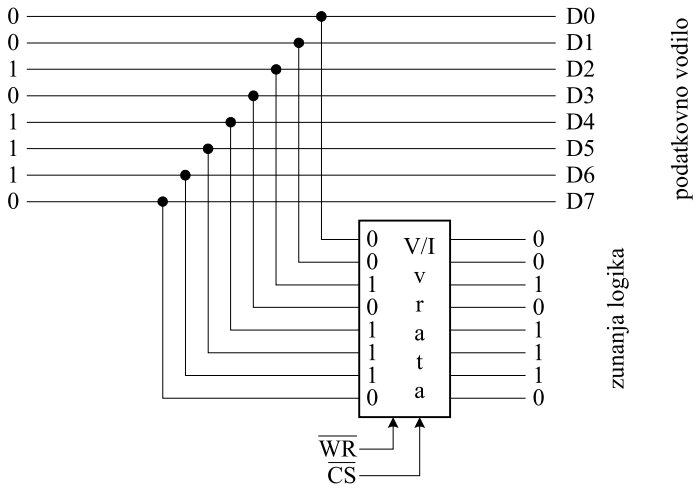
Prav tako pomemben je tudi prenos podatkov med CPE in V/I enotami. Pri tem moramo določiti vmesnik med mikroračunalniškim sistemom in zunanjo logiko, saj moramo zajeti možnosti prenosa podatkov, kot nadzor nad signali, ki zaznajo vsko spremembo na vhodni enoti. Prenos podatkov med perifernimi (V/I) enotami in lahko rešujemo na razne načine, vse pa lahko razporedimo v tri kategorije (slika 2.12):



Slika 2.12: Prenos podatkov

- programiran V/I. V tem primeru je prenos podatkov povsem pod nadzorom programa, ki se izvaja na CPE.
- prekinitveni V/I. Prekinitve generira zunanja logika in zahteva prekinitev izvajanja trenutno tekočega programa na CPE ter izvajanje programa (prekinitveve rutine), ki pripada aktivni prekinitvi.
- direktni dostop do pomnilnika (DMA - Direct Memory Access). Ta prenos se izvede direktno med pomnilnikom in aktivirano vhodno/izhodno enoto, brez posredovanja CPE, pod nadzorom posebne zunanje logike.

Vhodno/izhodna vrata predstavlja le V/I zunanji 8-bitni register, ki je povezan na podatkovno vodilo zunanjega sistemskega vodila (slika 2.8) in na ustrezno izbirno linijo iz perifernega dekoderja.



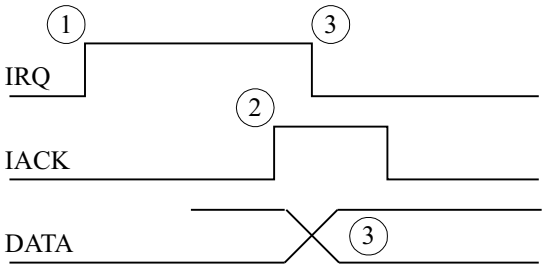
Slika 2.13: V/I vrata

V določenih sistemih se vhodno/izhodna vrata naslavljajo kot del pomnilniških lokacij, v drugih sistemih, pa je celoten prostor zunaj CPE razdeljen na pomnilniški podprostor in na periferni podprostor.

2.3.1 Prekinitve

Večina mikroprocesorjev ima kontrolne signale, preko katerih krmilniki perifernih enot sporočajo procesorju vse spremembe na perifernih enotah. Te kontrolne signale poznamo pod imenom *prekinitve* - *interrupts*.

Prekinitve nastopijo npr. pri spremembi ojačanja signala, kjer element za identifikacijo ojačanja zazna prekoračitev preko dopustnih toleranc. Ta element sproži signal zahteve za prekinitev (IRQ - *Interrupt Request Signal*), ki ga pošlje pošlje mikroprocesorju. Mikroprocesor lahko to zahtevo sprejme tudi zavrne. V primeru, da jo sprejme, se procesor na zahtevo po prekinitvi odzove s signalom - potrditev prekinitve IACK - (*Interrupt ACKnowledge*). Ta signal pošlje na zunanje sistemsko vodilo. Element ali naprava, ki je sprožila signal, uporabi potrditveni signal kot potrditev *enable* svoje zahteve po prenosu podatkov iz periferne V/I enote na sistemsko vodilo. Iz spodnjega diagrama je razviden potek posameznih signalov.



Slika 2.14: Zahteva in potrditev prekinitve

Seveda je naloga prekinitve v takojšni obdelavi programa procesa, ki je povzročil to prekinitev. To pa pomeni, da mora v primeru, ko je bil procesor zaseden z izvajanjem nekega programa, v trenutku potrditve prekinitve končati z izvajanjem tekočega programa in preiti na izvajanje prekinitvenega podprograma. Po zaključku izvajanja se procesor vrne na točko prekinitve osnovnega programa in nadaljuje z njegovim izvajanjem.

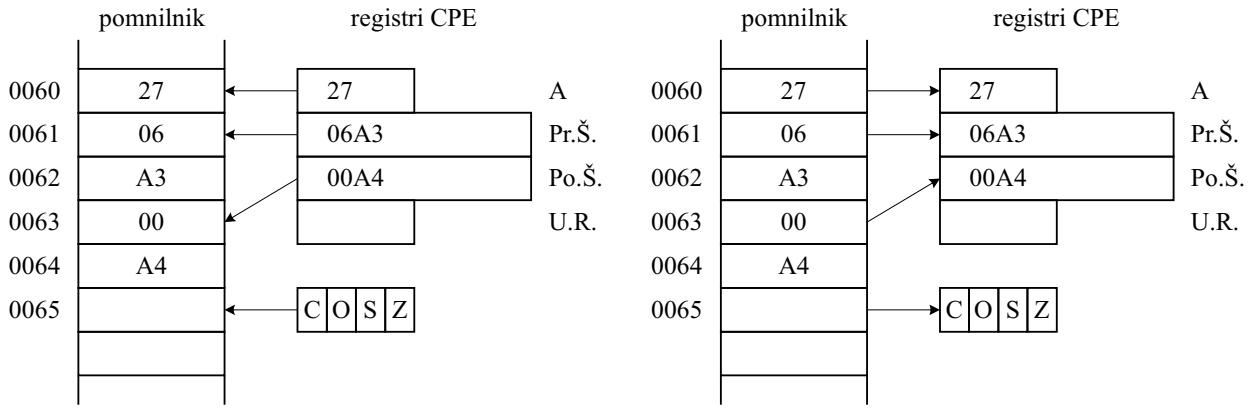
Odziv mikroprocesorja na prekinitev

Na najosnovnejšem nivoju se mikroprocesor odzove na prekinitev kar enostavno z zamenjavo vsebine programskega števca s vsebino začetnega naslova prekinitvi pripadajočega programa. Pri tem se takoj postavita dve vprašanji:

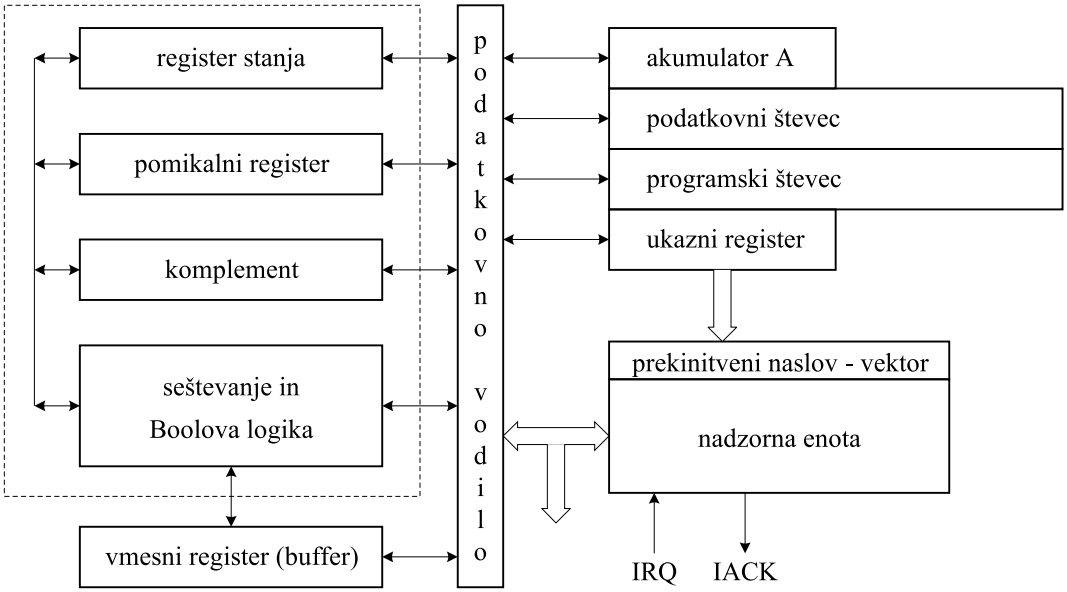
- kaj se zgodi s programom, ki se je izvajal in
- kje in kako mikroprocesor prebere naslov programa, ki se naj bi izvajal ob nastopu določene prekinitve.

Najprej si oglejmo, kaj se zgodi s programom, ki se je izvajal. Vse podatke registra stanja, akumulatorja in podatkovnega števca, moramo nekje shraniti, saj se ti izbrišejo takoj ko se prične izvajati nova rutina. Vsebine registrov moramo shraniti pred pričetkom izvajanja prekinitvenega programa. To omogoča obnovitev vsebine registrov po izvajanju prekinitvenega programa. Koncept je prikazan na sliki 2.15.

Kako pa mikroprocesor prebere ustrezen naslov programa, ki naj bi se izvajal na osnovi prekinitvene zahteve? Čim CPE sprejme prekinitveno zahtevo, tedaj takoj odgovori s signalom IACK, shrani vsebino vseh registrov na ustrezno lokacijo v pomnilniku in prebere vsebino (naslov programa) *prekinitvenega vektorja* v programski števec. Prekinitveni naslovni vektor se nahaja ali kot dodatni vektor v CPE, ali kot del zunanje logike z logiko vrat (ustrezni registri se nahajajo v samem V/I elementu).



Slika 2.15: Shranjevanje registrov v pomnilnik in nazaj



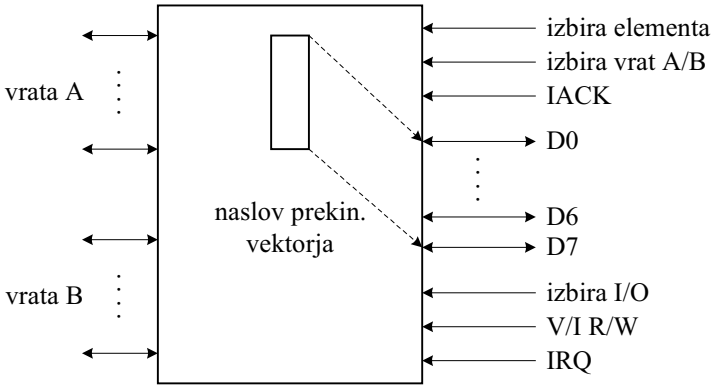
Slika 2.16: Naslov prekinitvenega vektorja

Predvidimo, da mikroprocesorska enota prevzame naslov programa, kateremu pripadajoča periferna logika povzroči prekinitev. V primeru ko nadzorna enota zazna prekinitveno zahtevo IRQ in le-ta dokonča trenutno izvajajoči ukaz v zbirniku, izvede sledeče operacije:

- generira potrditveni signal prekinitvene zahteve IACK,
- shrani vsebino registra stanj, akumulatorja, podatkovnega števca in programskega števca, ali dopusti da to opravi programer sam,
- premakne vsebino registra s prekinitvenim vektorjem (prekinitveni naslovni vektor) v register programskega števca.

V mnogih aplikacijah je smiselno vgraditi konstantne prekinitvene naslove, saj v komunikacijskem okolju procesor ne predstavlja del računalniškega sistema, pač pa del logične enote komunikacijskega sistema.

V vzporenih vhodno/izhodnih enotah nastopajo problemi s številom nožic, ki jih ima dotično VLSI vezje.

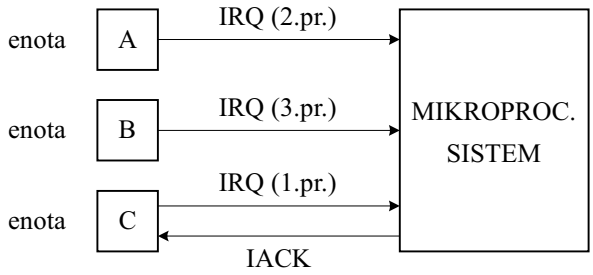


Slika 2.17: V/I enota z registrom prekinitvenega naslova

Prioritete prekinitev

Kaj se zgodi, ko v sistemu nastopa več V/I enot, ki zahtevajo prekinitev mikroprocesorja? V takem primeru je potrebno uvesti razlikovanje med prednostmi posameznih enot. Zato so uvedli posebno enoto za razvrščanje prioritet posameznih prekinitev. Nekateri procesorji že sami ločijo med prioritetami. Npr. procesor M6800 vsebuje dva tipa prekinitev. Eden ima višjo prioriteto pred drugim.

Na sliki 2.18 je razviden koncept prekinitvenih prioritet



Slika 2.18: Prekinitvene prioritete

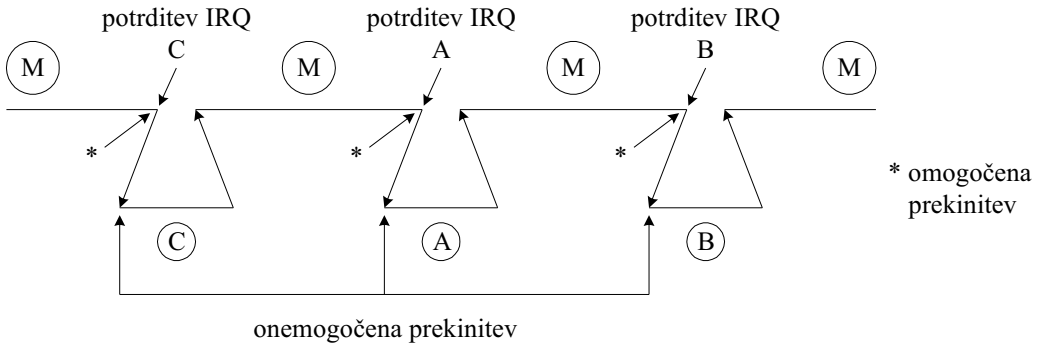
Vsaka od enot A, B in C zahteva prekinitev s signalom IRQ, poslanim mikroračunalniku. Vzemimo, da enota za določanje prioritet določi enoti C najvišjo prioriteto. Enota A ima drugo in enota B tretjo prioriteto (najnižjo). IACK potrditveni signal mora torej prejeti enota C.

Enoti A in B lahko sedaj umakneta zahtevo po prekinitvi ali pa ne. V slednjem primeru, se bo njihova zahteva po prekinitvi izvršila, ko se bo izvedel program, ki pripada višji prioriteti. Vzemimo torej tri pripadajoče programe, ki se nahajajo v pomnilniku računalnika. Programi se izvedejo na sledeči način, prikazan na sliki 2.19.

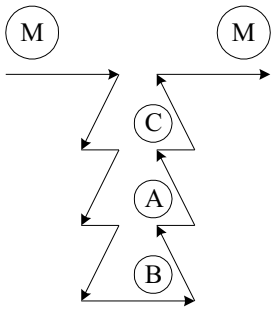
M predstavlja glavni program, A, B in C pa predstavljajo ustrezne prekinitvene programe.

Predpostavimo, da pa v trenutku, ko nastopi prekinitev, računalnik ne onemogoči nastopa novih prekinitev. V predhodnem primeru enot A, B in C se izvedejo ti programi na sledeči način, prikazan na sliki 2.20.

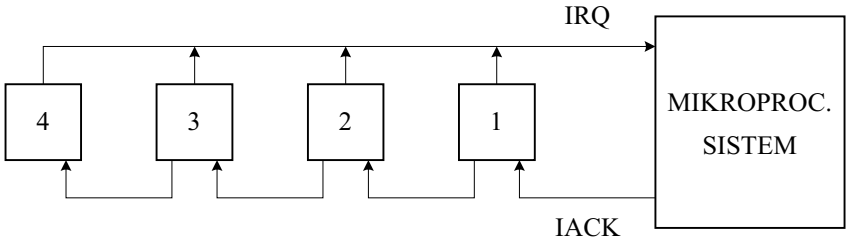
Najpomembneje je, da razumemo princip dodeljevanja IACK v primeru, ko se izvaja program B, tedaj ga ne more prekiniti nobena enota. Šele potem, ko se izvede program B, se lahko izvede program A itd. V primeru da pa mikroračunalnik ne vsebuje enote za razvrščanje prioritet, tedaj lahko periferne enote povežemo v nekakšno verigo (daisy chain). Več enot se v tem primeru poveže v verigo, ki je povezana z IACK linijo. Seveda morajo te V/I enote imeti vgrajeno interno logiko, ki prepusti signal IACK sosednji enoti v primeru, ko ta ni zahtevala prekinitve. Tako vse enote prepuste signal IACK, ki se ustavi pri enoti, ki je prva v verigi zahtevala prekinitev IRQ (slika 2.21).



Slika 2.19: Potek programov prekinitev



Slika 2.20: Potek programov prekinitev

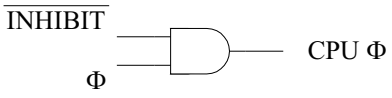


Slika 2.21: Potek programov prekinitev

Vsaka od omenjenih metod ima svoje prednosti in slabosti.

2.3.2 Direktni dostop do pomnilnika DMA

Veliko perifernih enot zahteva hiter prenos podatkov direktno v pomnilnik računalnika. V tem primeru zopet potrebujemo novo nadzorno enoto, ki omogoča direkten prenos podatkov z V/I vrat v spomin. Seveda CPE ne igra pri tem nikakršnega povezovalca med pomnilnikom in V/I enoto. Izključitev CPE iz tega prenosa je nujna, npr. z ustavitvijo urinega signala z dodatno logiko s slike 2.22. Signal INHIBIT je v stanju 0 aktiven in ustavi delovanje CPE.



Slika 2.22: Zaustavitev CPE z uro Φ

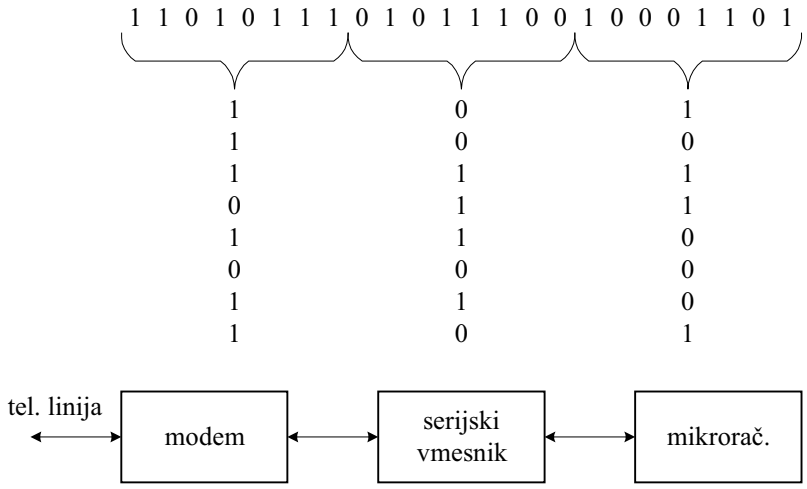
DMA enota mora vsebovati sledeče registre:

- naslovni register (kaže na naslednjo pomnilno lokacijo)
- števec, ki določa dolžino podatkovnega sklada
- register stanj, ki določa smer pretoka podatkov (ta tudi določa, če je DMA enota aktivna ali ne).

DMA enoti določimo na samem začetku izvajanja program vsebine vseh teh registrov. Prav tako mora DMA enota generirati ustrezne R/W krmilne signale. Čim startamo DMA enoto, se zunanja enota direktno priključi na podatkovno vodilo. Kot prekinitev imamo tudi tu linijo za postavitev zahteve DMA.

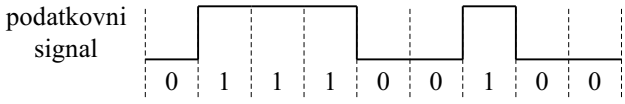
2.3.3 Serijski vhod/izhod

Podatki se po telefonski ali drugi komunikacijski poti prenašajo zaporedno – serijsko. V takem primeru mora CPE te serijsko podatke pretvoriti v vzporedne podatke, za kar potrebuje serijsko/vzporedni, oziroma vzporedno/serijski pretvornik.



Slika 2.23: Serijsko/vzporedni prenos

Vprašamo se, kako sprejemna enota sprejema serijske signale. Vzemimo pozitivno logiko, kjer je logična 1 predstavljena s $+5V$ in logična 0 z maso signala ter niz podatkov 011100100. Serijsko predstavitev podatkov vidimo na sliki 2.23.

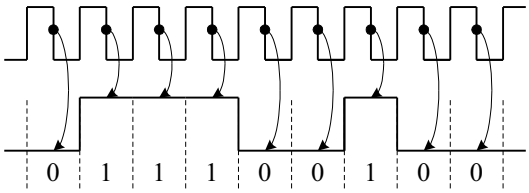


Slika 2.24: Serijski bitni signal

Urin signal uporabimo zato, da razpoznamo podatkov signal.

Vzemimo, da zadnja stran urinega signala omogoča identifikacijo trenutka, v katerem vzorčimo serijski signal, kot ga prikazuje slika 2.25. Seveda če sprejemnik uporablja signal ure za interpretacijo serijskega podatkovnega signala, potem mora tudi oddajnik uporabljati uro iste frekvence pri postopku pretvorbe in oddaje serijskega signala.

Dejansko vsak signal potrebuje nek čas za prehod iz enega v drugo stanje. Zato mora sprejemna enota čakati, da se serijski podatkovni signal ustali v nekem stanju. V te namene velikokrat uporabimo

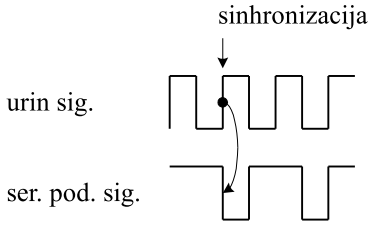


Slika 2.25: Branje serijskega signala

asimetrični urin signal, kjer oddajamo z zadnjo stranjo urinega signala in sprejemamo s prvo stranjo naslednjega signala.

Časovni interval, v katerem serijski podatkovni signal predstavlja en sam binarni zapis (digit) je v relaciji s hitrostjo prenosa podatkov. Hitrost prenosa merimo s številom bitov na sekundo.

Pri serijskem prenosu običajno uporabljamo mnogokratnik urinega signala, glede na podatkovni signal. Razlog za ta faktor ($\times$ 16, $\times$ 64) je v ideji, da moramo identificirati signal čimbolj na sredi njegovega intervala. Sprejemna enota lahko sama generira ustrezni urin signal, ki se sinhronizira z enim od prehodov serijskega podatkovnega signala (slika 2.26).



Slika 2.26: Prikaz generiranja urinega signala

Vidimo, da je sinhronizacija na bit enostavna, včasih pa tudi potrebujemo sinhronizacijo na "vzorec" (pattern). Temu vzorcu rečemo SYNC CHARACTER in se uporablja v sprejemnih enotah, kjer želimo zagotoviti korektno razlago oddanih podatkov.

2.3.4 Serijska V/I komunikacija

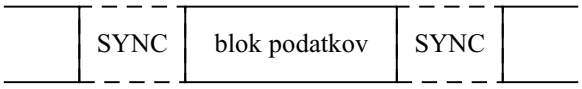
V splošnem lahko komunikacijske protokole razdelimo v dve kategoriji:

- sinhrone in
- asinhrone.

Sinhroni serijski podatkovni prenos je učinkovit in podatki se potrjujejo le ob točnem urinem signalu. Čim se določi hitrost prenosa, mora oddajna enota oddati bit vsak urin impulz, tako da sprejemna enota enolično razpozna serijski podatek. Sinhroni protokol tudi določa dolžino posameznih podatkovnih enot, s čimer omogoča, da se sprejemnik nekako sinhronizira na meje sinhronizacijskega intervala (enote). Zato se vsak podatkovni niz prične z enim ali dvema SYNC znakoma.

$$\overbrace{01101001}^{\text{SYNC1}}\overbrace{01101001}^{\text{SYNC2}}1011...$$

Podatkovni paket je v sinhronem serijskem podatkovnem nizu sestavljen iz samih podatkov, običajno brez bita parnosti. V fazi čakanja na podatke preide sprejemna enota v tako imenovani limitni način delovanja. V tem načinu naprava konstantno primerja prihajajoče vhodne podatke s SYNC znakom. Običajno upošteva dva SYNC simbola (sinhronizacija) in šele ko jih zazna, sprejemna enota začne s sprejemom podatkovnega paketa.

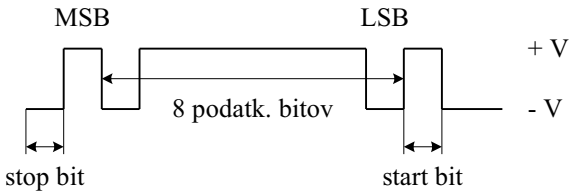


Slika 2.27: Sinhroni prenos

V tem primeru smo seveda predpostavili, da ima oddajna enota na razpolago podatke za oddajo. V primeru, da jih nima, tedaj bo oddajna enota zapolnila praznino s SYNC simboli, vse dokler nima na razpolago ustreznih podatkov. Tak način dela se uporablja v komunikaciji s tipkovnic, kjer je uporabnik običajno počasnejši od komukacijske enote.

Asinhroni prenos je nastal v začetnih fazah razvoja periferne opreme (TTY terminali). Terminal je npr. sprejel nekaj znakov iz računalnika. Sama naprava ne bi zaznala med posameznimi biti kjer se začne npr. znak za črko A itd. Zato smo namesto niza znakov obdali z ustreznimi sinhronizacijskimi znaki en sam "karakter".

Tako okvir sestavljata start in stop bit.



Slika 2.28: Organizacija asinhronega prenosa besede

V primeru, ko ni prenosa znakov, tedaj je linija na $-V(-12V)$. Čim pa se pojavi znak za prenos preide signal iz $-12V$ na $+12V$, za čas enega impulza, s čimer generira startni bit. Na koncu oddajnik odda stop bit tako, da drži linijo na $-V$ vsaj za čas trajanja enega bita.

Poglavje 3

Arhitektura komunikacijskih omrežij

3.1 Osnovni komunikacijski standardi in principi

V komunikacijskih sistemih vidimo, da je sporočilo, ki ga želimo prenesti, sestavljeno iz podatkov, nadzornega in sinhronizacijskega dela. Sporočila v sinhronih komunikacijskih sistemih vsebujejo sledeče elemente:

- uporabnikove podatke
- SYNC znake
- naslovno polje
- nadzorne znake
- ugotavljanje in popravljanje napak (slika 3.1).

Sporočilo - message - vsebuje glavo z naslovom in številko zaporednega niza in nekaj nadzornih polj.

SYNC	CONTR	ERROR CHECK	USER DATA	SEQUENCE NUMBER	ADDRESS	CONTR	SYNC	SYNC

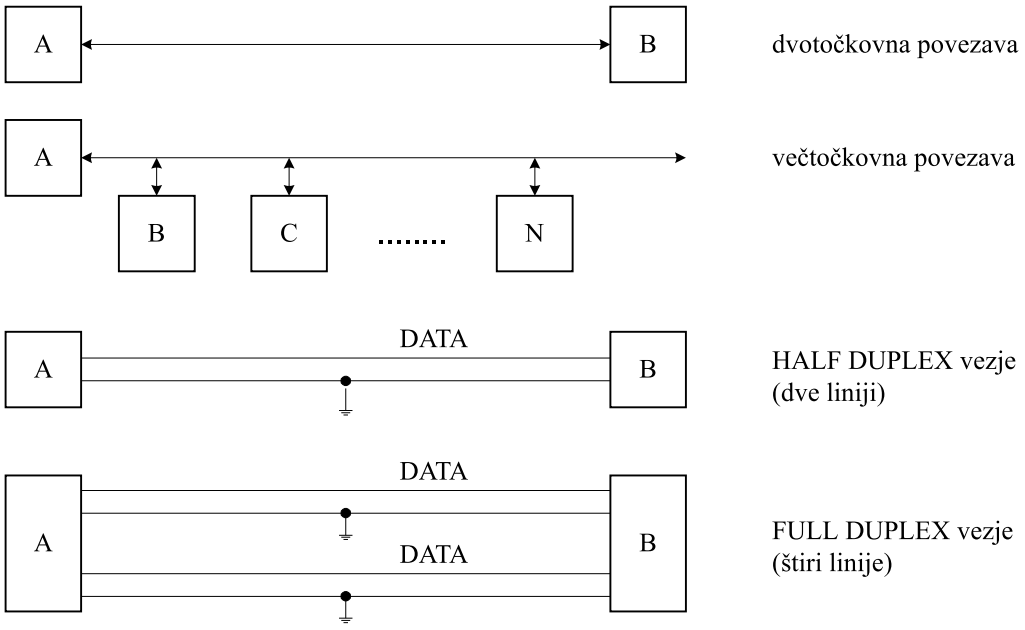
Slika 3.1: Organizacija sinhronega prenosa podatkovnega niza

Bitna sekvenca je tudi odvisna od uporabljene kode. Danes se najbolj uporabljata EBCDIC koda (Extended Binary Coded Decimal Interchange Code) in ASCII (American Standard Code for Information Inter...). EBCDIC je 8-bitna koda (IBM), dočim je ASCII 7-bitna koda. Prenosne poti omogočajo izmenjavo podatkov med posameznimi postajami. Električne lastnosti tega medija vplivajo na lastnosti, kot na izgradnjo komunikacijskega sistema. V bistvu poznamo sledeče tipe poti (slika 3.2):

- točka - točka in večtočkovne (multidrop) povezave,
- simplex, polovični duplex, duplex,
- najete linije, linije preko central.

3.1.1 Vmesniški standardi

RS232-C je standard ameriškega združenja elektronske industrije EIA. CCITT je objavil podobna standarda:



Slika 3.2: Osnovni primeri povezav

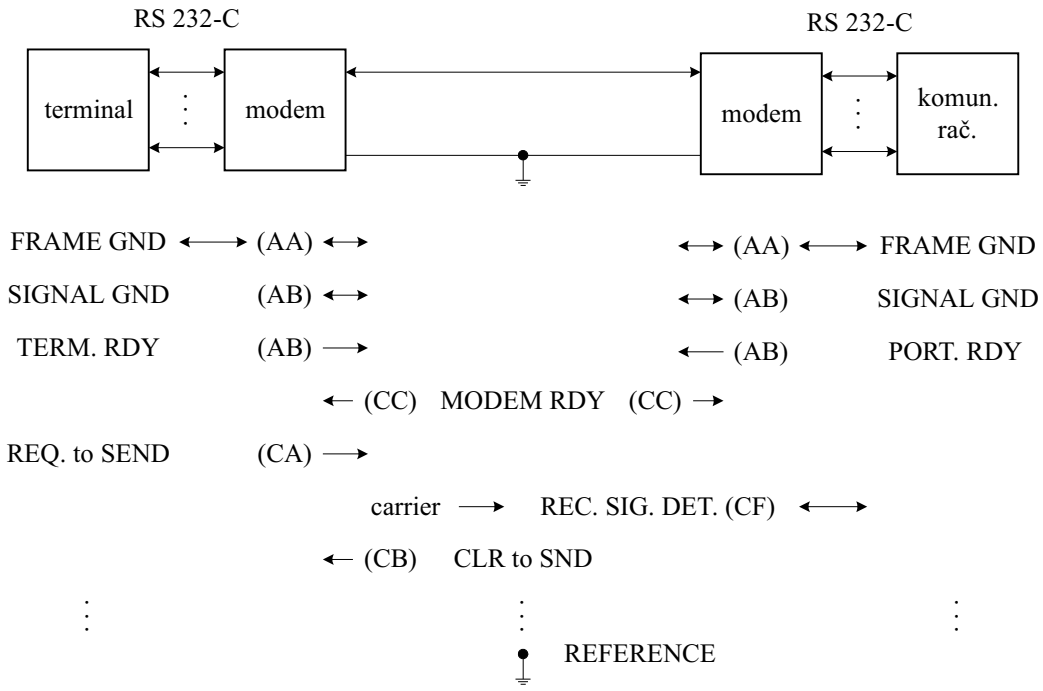
- V.24, ki opisuje lastnosti vezja in
- V.28, ki opisuje električne lastnosti vezja.

Vmesnik omogoča prenos podatkov, nadzor signalov po komunikacijski poti, sinhronizacijo in hitrost prenosa ter maso električnega signala. Vezje se zaključi z običajnem 25 "pin"-skem komunikatorju. Poleg standardnih priključkov vsebuje še dodatne signale, ki omogočajo priključitev vezja ma modem (modulator/demodulator), ki omogoči preslikavo (prenos) signala iz digitalnega v analogni svet. Pretok podatkov preko RS232-C vmesnika je prikazan na sliki 3.3. Uporaba "half/full duplex" ali najetih/preklopnih linij določa način aktiviranja vezij. Slika 3.3 prikazuje privatno "half duplex" povezavo.

- RS449 je izboljšani RS232-C standard. RS232 je omejen na 20 kb/sec in je priporočljiv le za male razdalje med komponentama sistema. RS449 pa vsebuje 37 osnovnih vezij poleg 10 dodatnih in ostalih testnih vezij. Standard je kompatibilen s CCITT in ISO standardi in omogoča prenos do 2 Mb/sec na večjih razdaljah.

RS449 standard omogoča opuščanje RS232 standarda. Med obema obstajajo adapterji, ki prilagajajo en standard k drugemu in obratno. Kljub hitri razširitvi standarda v obliki RS422 pa se klasični RS449 ni uveljavil iz več razlogov:

- veliko vlaganja v RS232-C
- večji konektorji (več priključkov)
- veliko aplikacij ne potrebuje velikih hitrosti
- večji stroški
- razvoj in pojav standarda ISDN (Integrated Services Digital Network)
- izboljšava RS232-C v RS232-D



Slika 3.3: "Half duplex" povezava terminala z računalnikom

3.2 Tehnika prenosa - protokoli

Postopke prenosa podatkov razvrstimo v pet skupin:

- asinhrone (start-stop) postopke,
- sinhrone znakovne postopke za povezavo točka - točka (karakter)
- sinhrone znakovne postopke za povezavo več točk (karakter- znak),
- sinhrone bitno orientirane postopke (stikalne centre - centrale) za možnost iskanja povezav po mreži,
- sinhrone bitno orientirane postopke za paketno preklapljanje.

Prve štiri skupine lahko zasledimo v:

GPD (General Purpose Disciplin) enostaven START/STOP postopek z uporabo BCD kode.

STR (Sinhronon Transit/Receive) sinhron postopek z uporabo SYNC znaka. Prenos in krmiljenje sta ločena postopka. Naslavljanje omejuje uporabo povezave tipa točka - točka, pri čemer se lahko uporabljata "half" in "full duplex" prenos.

BSC binarno sinhrona komunikacija je univerzalen sinhron protokol za fiksne in poljubno nastavljive (izberljive) povezave, povezave točka - točka in za večtočkovne povezave kot so nizi in delo v dialogu. Osnovni znaki so kodirani v EBCDIC ali ASCII kodi, protokol je znakovno in ne bitno usmerjen in deluje v "half duplex" načinu obratovanja. Vsebuje poseben postopek za odpravljanje napak. Postopek je primeren za obdelavo v nizih, dočim pri delu z uporabo dialoga med terminali pa ni učinkovit.

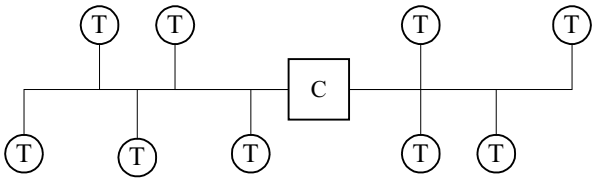
SDLC (Synchronous Data Link Control) To je sinhron bitno usmerjen postopek, ki vsebuje transparentno kodiran podatkovni prenos. Za razliko z BSC načinom ima SDLC način celovit nadzor in blokovno preverjanje vgrajeno v protokolu. Postopek omogoča oštevilčenje podatkovnih blokov, blokovno oblikovanje v okvirje (frames) prenos po dupleksnem načinu za povezave točka - točka, kot tudi za večtočkovne povezave.

Za zadnjo metodo veljajo posebna CCITT priporočila. Ta določajo naslednji način povezave, kot je:

X-25 je sinhron in bitno usmerjen protokol. Posebej je prirejen za paketno delo. Nastavitev fiksnih linij omogoča pri X-25 delo z vertikalnimi povezavami. Podatkovni paket se v bistvu pošlje v vozlišče mreže, nakar se transportira po poljubni poti naslovljenemu sprejemniku. Vsebina paketa je pri tem poljubna, kot je tudi poljubna pot znotraj omrežja.

3.2.1 Tehnika zasnovana na sporočilih

Kot smo spoznali so omrežja, ki temelje na prenosu znakov sestavljena iz neinteligentnih terminalov in iz centralnega računalika. Najenostavnejši sistemi, zasnovani na izmenjavi sporočil, vsebujejo kot osrednjo enoto en sam računalnik. Torej v principu ločimo dva tipa naprav; prve so namenjene omrežjem, zasnovanem na izmenjavi sporočil ter tiste za oddaljeno (remote job entry) komuniciranje.



Slika 3.4: Povezava terminalov z računalnikom z uporabo sporočil

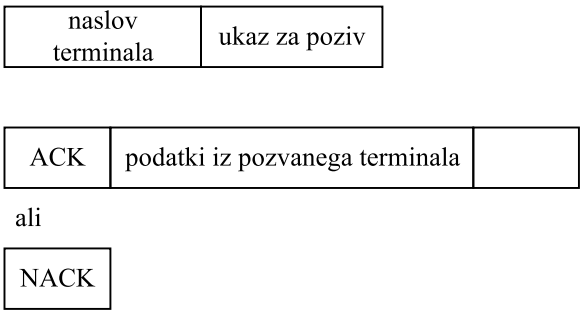
3.2.2 Multipoint - (multidrop) sistemi

Tovrstno omrežje omogoča optimalno uporabo prenosnega medija med več nanj priključenimi terminali. Prav tako lahko več komunikacijskih poti priključimo na osrednji računalnik. Vsak terminal s slike 3.4 lahko zahteva priključek na PSTN (Public Switch Telephon Network) preko modema. V primeru, ko imamo na voljo le en kanal, tedaj lahko prenašamo le eno sporočilo v danem trenutku (med računalnikom in terminalom). Zato, da se izognemo medsebojnim vplivom, se zahteva disciplina pri prenosu sporočil na samem komunikacijskem kanalu. Za to skrbi nadzorna enota, ki je del računalnika. Sporočila se prenašajo s pozivanjem terminalov (polling), če žele prenašati sporočila. Vsak terminal lahko sprejema vsa sporočila na kanalu, zato mora sporočilo, ki ga pošilja računalnik, vsebovati naslov prejemnika.

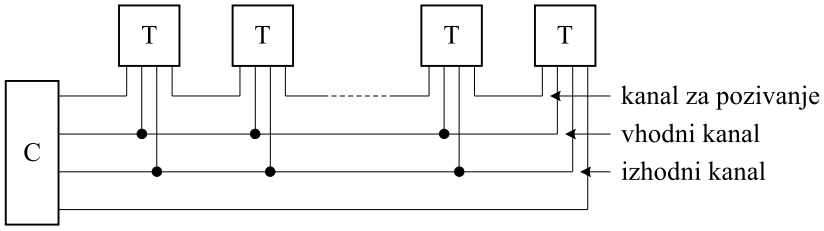
Ker se morajo podatki najprej nabrati v terminalu ali računalniku, kar omgoča večjo koristno uporabnost prenosnega kanala, je za ta način primeren sinhron znakovno organiziran okvir za prenos po kanalu.

V tem primeru sta znana dva načina pozivanja. Prvi je s klicanjem “Call Polling” in drugi je pozivanje v multipoint sistemu imenovanem “Hub Polling”. Prvi način je prikazan na sliki 3.5. V drugem primeru se pozivanje prenaša s terminala na terminal, ki so vezani v sistem (sekvenčen način pozivanja). Tudi če se je oglasil nek terminal s podatki, se pozivanje nadaljuje od tega terminala dalje vse do zadnjega terminala.

Seveda so pri izbiri ustrezne organizacije pozivanja pomembni čas odziva, učinkovitost, fleksibilnost, cena opreme.



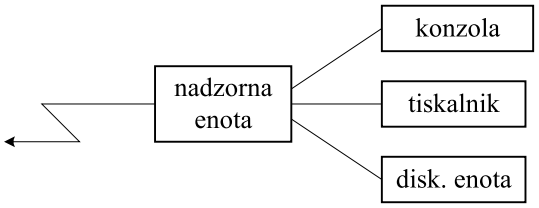
Slika 3.5: Pozivanje terminala



Slika 3.6: Tehnika komuniciranja z uporabo sporočil

3.2.3 Oddaljene postaje (Remote Job Entry Stations - RJES)

Ta način dela je namenjen predvsem delu v ozadju, delu z manjšim številom terminalov. Osnovna ideja je zrasla na potrebi oddaljenih uporabnikov, ki so hoteli imeti poleg sebe še ustrezno periferijo s terminalom. RJES se danes še uporabljajo kot periferni industrijski terminali. Prenos je znakovno organiziran tako, da se v resnici med računalnikom in periferno enoto prenaša podatkovni blok. Če se znaki prenašajo v blokih, tedaj RJES uporabljajo komunikacijo s sporočili po sinhronem mediju. V tem primeru se na isti liniji lahko nahajajo RJES kot terminali.



Slika 3.7: Koncept oddaljene postaje

Koncept oddaljene postaje se je spremenil čim so se pojavili mali osebni računalniki, tako da že lahko govorimo o razdeljenem računanju.

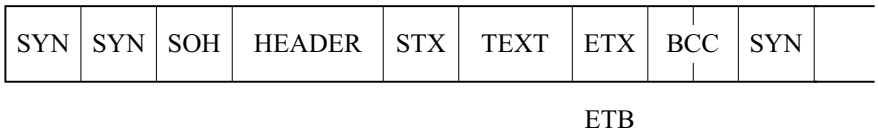
3.2.4 Organizacija enostavnega sporočila (BSC)

S sporočili dejansko v večtočkovnih, kot v RJE povezavah, nadziramo izmenjavo blokov podatkovnih znakov. Uporabljen protokol je v tem primeru zelo pomemben. Najčešče se uporablja že omenjeni BSC - Binarno Sinhroni Komunikacijski protokol (IBM). Ta določa postopek izmenjave nadzornih sporočil tako, da poteka prenos podatkov med postajama, ki sta povezani s komunikacijskim medijem.

BSC ima tudi nekaj podvariant. Osnovna organizacija oblike sporočila je prikazana na sliki 3.8

STX, SYN, ETX in ETB so posebni znaki za nadzor, ki razmejujejo različne dele sporočila.

SOH (Section Over Head) pove, da je HEADER vključen v sporočilu



Slika 3.8: Oblika sporočila BSC

STX (Start of teXT) pove, da je sledeči znak del teksta sporočila ali podatkov. Tekstovno polje je točno določeno s številom znakov, ki jih lahko vsebuje.

ETX (End of teXT) označuje konec teksta uporabnikovega sporočila. V primeru, da imamo preveč podatkov, ki bi jih želeli prenesti, a jih ne moremo prenesti v enem samem bloku, tedaj v naslednjih blokih ne uporabimo ETX znaka, ampak ga uporabimo le v zadnjem prenešenem bloku.

ETB (End of Transmission Block) tega uporabimo namesto ETX v primeru, ko je blok podatkov večji od dolžine BLOKA BSC FORMATA. Ta znak pomeni le konec podatkov v določenem prenešenem bloku.

BCC (Block Chech Character) pomeni preverjanje pravilnosti sporočila, ki ga je računal tudi oddajni računalnik (terminal). Večina sistemov vsebuje izračun parnosti sporočila.

Operacija BSC sporočila zahteva, da oddajnik kot sprejemnik:

- omogočita povezavo - oddajnik mora vedeti, da je sprejemnik pripravljen na sprejem,
- zanesljiv prenos podatkov - brez napak ali izgube
- zaključek prenosa tako, da sprejemnik ve, da naj ne pričakuje novih podatkov.

3.2.5 SDLC sinhroni protokol

Ta danes predstavlja osnovo večine podatkovnih komunikacijskih protokolov. IBM ga je uvedel 1973. leta za delo v bančnih sistemih. O njem govorimo tudi zato, ker je postal osnova HDLC sistema.

SDLC uporablja sinhrono, bitno usmerjeno, "Go-back" metodo. Uporabljamo ga lahko v enostavnih linijskih povezavah, kot je točka - točka, v večtočkovnih povezavah in v mrežnih zankah. Protokol deluje na half-duplex, preklopnih in privatnih (najetih) linijah.

SDLC lahko tudi uporablja kombinacije med metodo pozivanja in metodo naslavljanja. Pozivajoča postaja - primarna prične s protokolom in je odgovorna za potek komunikacije s pozvano sekundarno postajo. V kontekstu sklopa večih mrež je lahko primarna postaja v eni mreži, sekundarna postaja pa v drugi mreži. SDLC sporočila se prenašajo v posebno predpisanem okvirju (formatu). Začetni in končni znak sestavlja 8 bitov 01111110 (flag) oziroma nekakšen SYNC znak.

Slika 3.9: Oblika sporočila SDLC

Zaključni FLAG lahko predstavlja tudi začetni SYNC znak za naslednje sporočilo.

SDLC je kodno jasno zasnovan in ne dopušča nastop niza bitov 01111110 iz FLAG polja. V primeru takega zaporedja podatkov, kontrolna logika obvezno vrne po petih 1 še 0. Na sprejemni strani se vedno odstrani 0, ki sledi zaporedju petih 1. Ker ni znakovno usmerjen, dolžina polja ni vedno mnogokratnih

8-bitov. Naslovno polje, ki sledi začetnemu FLAG polju, določa enoto, ki jo naslavljamo. SDLC omogoča tudi skupinsko naslavljanje kot tudi globalno naslavljanje vseh postaj (Boadcast). Odziv postaje vedno vsebuje tudi njen naslov.

Nadzorno polje določa funkcijo celotnega sporočila in vpliva na logiko na sprejemni in oddajni strani. To polje je 8-bitno in je lahko določeno v naslednjih oblikah:

(U) (Unnumbered) neoštevilčen format - oblika sporočila se uporablja za nadzor, inicializacijo postaj, odklop, testiranje in nadzor nad odzivi posameznih polj.

(S) (Supervisor) - nadzorni format se uporablja za potrjevanje ACK ali za zanikanje NACK informacijskih sporočil. Nadzorni format ne vsebuje uporabniških podatkov. Dejansko z njim potrjujemo sprejem podatkov, pogoje "busy" - zaseden, "ready" - pripravljen in sporočamo številko napačno sprejetega sporočila.

(I) (Information) - podatkovni prenos vsebuje uporabniške podatke.

FCS - (Frame Check Sequence Field) vsebuje 16-bitni podatek, ki se izračuna na osnovi vsebine naslovnega, nadzornega in podatkovnega polja, na strani oddajne postaje.

Nadzorno polje pri SDLC prenosu

Biti z najnižjo utežjo (4 desni biti) določajo obliko prenešenega okvirja sporočila (11 - neoštevilčenih, 10 nadzornih, 0 informacijskih). Ostanek bitov določa poseben tip funkcij v okviru sporočila.

Command Response	Format		Low Order		Command	Response	I-Field Prohibited	Resets Nr and Ns
UI	U	000	P/F	0011	×	×		
RIM	U	000	F	0111		×	×	
SIM	U	000	P	0111	×		×	×
SNRM	U	100	P	0011	×		×	×
DM	U	000	F	1111		×	×	
DISC	U	010	P	0011	×		×	
UA	U	011	F	0011		×	×	
FRMR	U	100	F	0111		×		
BCN	U	111	F	1111		×	×	
CFGR	U	110	P/F	0111	×	×		
RD	U	010	F	0011		×	×	
XID	U	101	P/F	1111	×	×		
UP	U	001	P	0011	×		×	
TEST	U	111	P/F	0011	×	×		
RR	S	Nr	P/F	0001	×	×	×	
RNR	S	Nr	P/F	0101	×	×	×	
REJ	S	Nr	P/F	1001	×	×	×	
I	I	Nr	P/F	Ns 0	×	×		

Tabela 3.1: SDLC Nadzorno polje

UI (Unnumbered Information) Neoštevilčena informacija služi za prenos podatkov uporabnika v neoštevilčenem okvirju

RIM (Request Initialization Mode) Inicializacija - start. V RIM okvirju sporočila zahteva sekundarna postaja od primarne postaje sprejetje SIM ukaza.

SIM (Set Initialization Mode) Izvedi inicializacijo. Odziv je UA.

SNRM (Set Normal Response Mode) Postavitev normalnega odziva postavlja sekundarna postaja v način normalnega odzivanja. Pri tem ime primarna postaja ves nadzor na povezavami.

DM (Disconect Mode) To sporočilo pošilja sekundarna postaja primarni postaji, da se želi izključiti iz obratovanja.

DISC (DISConect) Ta ukaz s primarne postaje potrdi zahtevo sekundarne postaje in jo postavi v odklopljen način delovanja (npr. kot da bi obesili slušalko).

UA (Unnumbered Acknowledgement) To je potrjevanje - ACK na SNMR, DISC ali SIM ukaz.

FRMR (FRaMe Reject) Zavrnitev sporočila polje skundarna postaja tedaj, ko sprejme neveljavno sporočilo (nepravilna številka).

BCN (BeaCoN) Signal, svarilo, signalizacija je uporabljen v komunikacijski zanki.

RD (Request Disconect) Zahteva sekundarne postaje se izloči.

XID (eXchange station IDentification) Zahteva identifikacijo sekundarne postaje. Uporablja se v sistemih s preklapljanjem.

UP (Unnumbered Polls)

Test Odloča, izvaja o testnih odzivih sekundarnih postaj.

RR (Receive Ready) Označuje, da so primarne ali sekundarne postaje pripravljene na sprejem.

RNR (Receive Not Ready).

REJ (REJect) To sporočilo eksplicitno zahteva ponovitev prenosa določenega sporočila.

I (Information) To sporočilo vsebuje uporabnikove podatke.

3.2.6 Nadzor pretoka sporočil

SDLC uporablja tehniko pomikajočega okna pri upravljanju protokola med oddajnikom in sprejemnikom. Vsaka postaja vsebuje sprejemni Nr in oddajni Ns števec.

Oddajna enota šteje vsako odposlano sporočilo, hkrati pa tudi pošilja stanje števca Ns v nadzornem delu sporočil. Sprejemna enota sprejme in verificira sporočila.

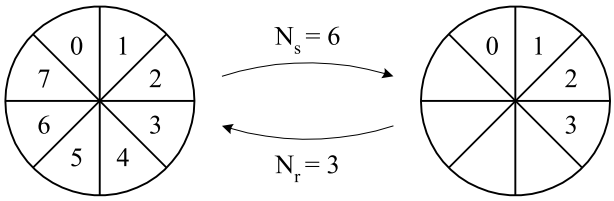
Če je sporočilo brez napak in ima ustrezno zaporedno številko Nr, tedaj pošlje signal ACK z ustreznim Nr, s katerim potrjuje sprejem sporočila Nr. Na koncu celotnega prenosa morata biti števca Ns in Ns enaka. Pri SDLC je števec 3-bitni, torej je dolžina Ns ali Nr največ 7. V primeru daljših sporočil pričenjamo šteti z 0.

Bit P/F (poll and final) je peti bit v nadzornem polju, ki

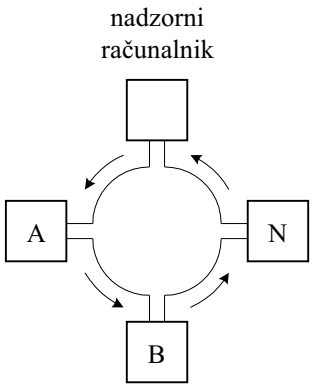
- ga primarna postaja postavi v P, kar pomeni pozivanje sekundarne postaje;
- ali ga sekundarna postaja postavi v F, kar pomeni, da se prenaša zadnje sporočilo.

Torej P bit zahteva odziv sekundarne postaje v obliki, ko je F bit 1. (P bit je vsebovan v sporočilu primarne, F bit pa v sporočilu sekundarne postaje).

Prenos v zanki je prav tako mogoč pri SDLC. Pri tem prevzame primarna postaja nadzor nad zanko. Vsaka sekundarna postaja dekodira naslovno polje vsakega sporočila. Tega lahko sprejme ali zavrne. Sporočilo se nato prenese do naslednje postaje v zanki.



Slika 3.10: Potrjevanje okvirjev pri pretoku sporočil



Slika 3.11: Sporočilo z žetonom

Čim nadzorni računalnik dokonča prenos ukaznega ali nadzornega sporočila, pošlje vsem zaporedni 0, s čimer signalizira sekundarnim postajam, da je zaključil s sporočili. Nato prične s konstantnim oddajanjem (1) enic, s čimer sporoča, da se je postavil v stanje sprejema.

SDLC ukazi so podmnožica HDLC ukazov, pri čemer SDLC deluje v HDLC-jevem neuravnoteženem načinu dela. HDLC vsebuje tudi uravnotežen (balanced) način dela, ki je primernejši za povezave točka - točka.

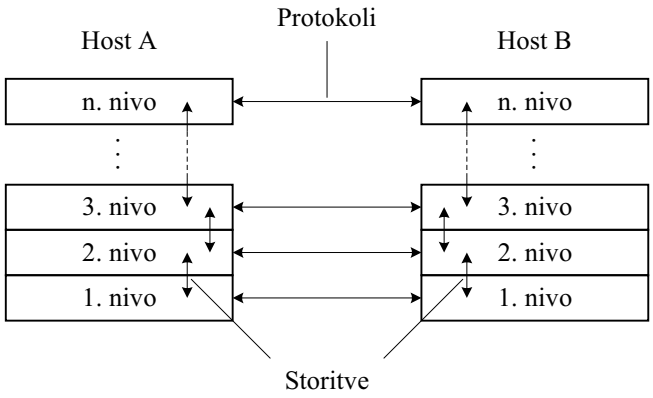
Neuravnotežen način pa je primeren za večtočkovne povezave (High Level Data Link Control).

3.3 Komunikacijski protokoli

3.3.1 Večnivojski komunikacijski protokoli

S pojmom arhitektura komunikacijskih omrežij označujemo vso programsko in strojno opremo, ki je potrebna za nemoteno komunikacijo med dvema ali večimi komponentami računalniškega omrežja. Vsi omrežni komunikacijski sistemi danes zaradi lažje praktične izvedbe ter nenazadnje zaradi geografske porazdeljenosti uporabljajo večnivojske protokole, ki bistveno zmanjšajo kompleksnost povezav posameznih vozlov v omrežju. Tovrstni protokoli z uporabo nivojev omogočajo relativno enostaven in predvsem od proizvajalcev strojne opreme neodvisen razvoj velikega števila novih mrežnih komponent. Ker so posamezni nivoji komunikacijskih protokolov popolnoma zaključene enote, dopuščajo enostavno fizično uporabo nivojev na različnih geografskih lokacijah.

Pri prenosu podatkov s pomočjo večnivojskih protokolov komunikacijski sistem dejansko pošlje podatke skozi posamezne nivoje (slika 3.12), ki sprejetemu bloku v glavo dodajo vsaj svoje naslovne informacije, pogosto pa tudi druge funkcije, ki pripadajo dotičnemu protokolu. Te so najpogosteje namenjene zaščiti podatkov pri prenosu preko komunikacijskega kanala. Po prehodu skozi vse nivoje se tako spremenjeni blok prenese po omrežju od pošiljatelja k sprejemniku, kjer se v obratni smeri ponovi postopek iz oddajne strani. Vsak nivo sprejetemu bloku sname pripadajoče naslovne podatke in opravi ostala preverjanja pravilnosti sprejetega bloka.



Slika 3.12: Splošni večnivojski komunikacijski protokol

Pri uporabi večnivojskih komunikacijskih protokolov so torej uporabniki storitev posameznega protokola vedno protokoli, ki v hierarhični razvrstitvi leže nad njim, tako da s stališča posameznega protokola na oddajni strani obstaja v času medsebojne komunikacije med dvema enotama logična povezava z dotičnim protokolom na sprejemni strani komunikacijskega kanala (slika 3.12).

3.3.2 Komunikacijski protokoli s spojem in brez spoja

Komunikacijski protokoli, ki zagotavljajo storitve prenosa podatkov, se v splošnem pojavljajo v dveh oblikah. Lahko zagotavljajo storitev s spojem (connection oriented service) ali storitev brez spoja (connectionless oriented service).

Komunikacijski proces, ki uporablja protokol s spojem, se sestoji iz treh faz: vzpostavitve zveze, prenosa podatkov in prekinitve zveze. Pri protokolu s spojem prenos podatkov vedno vključuje tri strani: pošiljatelja, prejemnika in komunikacijski protokol. Če hoče pošiljatelj poslati sporočilo večim prejemnikom, mora postopek ponoviti za vsakega prejemnika posebej. Pri takem načinu prenosa podatkov mora biti prejemnik popolnoma identificiran v času, ko je zveza vzpostavljena. Ves čas, dokler je zveza vzpostavljena, namreč pošiljatelj lahko domneva, da je sporočilo uspešno preneseno in da so podatki sprejeti v istem vrstnem redu, kot so bili poslani. Če pride na eni izmed treh strani vključenih v zvezo do napake, se zveza prekine in obe strani sta o tem obveščeni. Zaradi tega se tak prenos podatkov obravnava kot zanesljiv prenos.

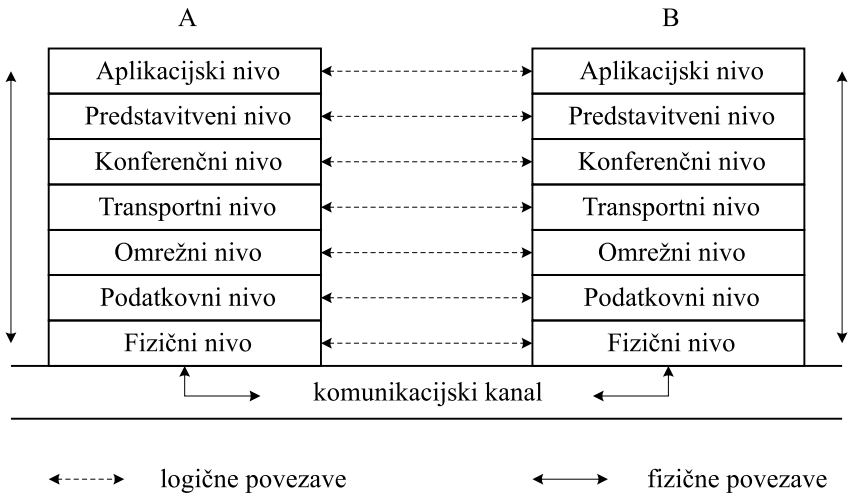
Protokol brez spoja deluje podobno kot poštni sistem. Proces, ki uporablja protokol brez spoja, sprejme vsako sporočilo in ga odpošlje podobno, kot poštni sistem sprejema naslovljena pisma in jih odpošilja prejemnikom. Pri protokolu brez spoja poteka prenos v eni fazi in pri tem dejansko ne pride do fizičnih povezav med pošiljateljem in prejemnikom. Pri tem načinu prenosa na fizičnem nivoju ni uporabljen noben mehanizem, ki bi zagotavljal skladnost med vrstnim redom oddanih in sprejetih sporočil ter preverjal in odpravljal morebitne napake, do katerih je prišlo med samim prenosom. Pošiljatelj pri uporabi protokolov brez spoja s strani prejemnika prav tako ne dobi nikakršnega potrdila o uspešnem sprejemu sporočila.

3.4 ISO-OSI model

ISO (International Standard Organization) je leta 1977 začel razvijati večnivojski komunikacijski OSI (Open System Interconnection) model, katerega poglavitni cilj je bil določiti usluge posameznih nivojev in tako razvijalcem standardov posameznih nivojev pustiti pri svojem delu povsem proste roke.

3.4.1 Funkcijski nivoji ISO-OSI modela

Fizični nivo je odgovoren za prenos električnih, optičnih ali radijskih signalov med dvema enotama, ne da bi prenešene signale kakorkoli interpretiral. Njegova naloga je v aktiviranju, oskrbovanju in zaključku fizičnih povezav med posameznimi enotami komunikacijskega omreža; najpogosteje med terminali DTE - (Data Terminal Equipment) in sistemskim nadzornim vezjem DCE - (Data Circuit-terminator Equipment).



Slika 3.13: ISO-OSI model

V praksi dokaj pogosto uprabljani protokoli iz fizičnega nivoja so, poleg že omenjenih protokolov RS232-C in RS449, protokoli X.21 in X.21 BIS ter V.24.

Podatkovni nivo je v OSI modelu zadolžen za zagotavljanje prenosa podatkov med pošiljateljem na eni in prejemnikom na drugi strani. S pomočjo svojstvenih nadzornih mehanizmov podatkovni nivo prenaša podatke v skupinah, imenovanih okvirji (frames).

Najpogostejša protokola iz tega nivoja sta SDLC in njegov naslednik HDLC.

Omrežni nivo je zadolžen za posredovanje in usmerjanje podatkovnih blokov, pogosto imenovanih paketov, skozi omrežje. Paket, ki ga transportni nivo pošiljatelja pošlje skozi omrežje, lahko tako prepotuje poljubno število najrazličnejših vmesnih sistemov, ne da bi se transportni nivo sprejemnika tega sploh zavedal.

Transportni nivo je ISO določil šele leta 1984 in predstavlja enega najbolj kritičnih delov OSI modela omrežja, saj je le-ta prvi nivo, nad katerim ima končni uporabnik popolni nadzor. Transportni nivo zagotavlja takšen tip povezave med dvema uporabnikoma, ki se najbolje prilega uporabnikovim zahtevam po kvaliteti, ceni, preklapljanju, preslikavi naslovov itd. Pomembna funkcija transportnega nivoja je tudi odkrivanje in odpravljanje napak, do katerih je prišlo med prenosom paketov. ISO standard je po zmožnosti odkrivanja in odpravljanja napak razdelil transportni nivo na pet razredov.

Konferenčni nivo je namenjen predvsem posameznim nalogam organizacije in sinhronizacije posameznih dialogov in upravljanju izmenjav podatkov med posameznimi komponentami mrež. Nivo seje omogoča:

- upravljanje interakcij
  - dvosmernih simultanih povezav, pri katerih programi lahko hkrati oddajajo in sprejemajo,

  - dvosmernih alternativnih povezav, pri katerih programi izmenoma oddajajo in sprejemajo,
  - enosmernih povezav, pri katerih en program samo oddaja in drugi samo sprejema.
- izvajanje prioritet in normalnega posredovanja podatkov,
- izvajanje uslug omogoča pošiljanje zahteve nivoju predstavitve, da se zadrži tekočo predstavitev, dokler se ne sprostijo vsi potrebni podatki.

Predstavitveni nivo omogoča pretvarjanje, formatiranje in sintaksno obdelavo znakov in kod, saj npr. množica najrazličnejših računalnikov in druge terminalske opreme, ki jo je moč dandanes povezati v enotno omrežje še zdaleč ne uporablja enakega kodiranja vseh tipov podatkov. Obstajajo trije protokoli nivoja predstavitve. Prvi je protokol virtualnega terminala, ki omogoča simulacijo različnih tipov terminalov in razvoj najrazličnejših aplikacij zanje. Drugi je virtualni datotečni protokol, ki je namenjen delu z datotekami in komunikaciji med njimi. Tretji protokol pa omogoča prehajanje posameznih nalog med posameznimi mrežnimi komponentami in zapis ustreznih struktur.

Aplikacijski nivo je namenjen uporabnikom, katerim omogoča:

- storitev dostopa do nižjih nivojev OSI modela, ki ni odvisen od narave uporabnikove aplikacije,
- storitev paketnega prenosa podatkov, dostop do podatkovnih baz in oddaljeno izvajanje nalog (RJE - Remote Job Entry) in
- storitev, ki služi podobnim aplikacijam, kot so verifikacija kreditnih kartic, trgovinskih blagajn itd.

3.5 X.25 protokol

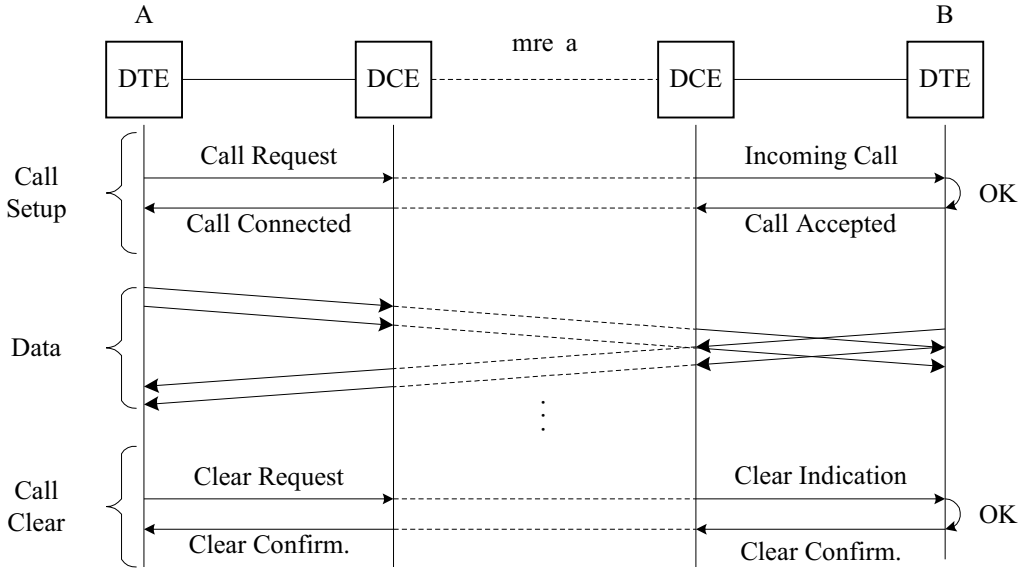
Komunikacijski protokol X.25, ki je bil s strani CCITT razvit kot standardni vmesnik za izvajanje uslug paketnega omrežja (PSDN - Packet Switched Data Network) in se vse pogosteje uporablja za povezovanje lokalnih mrež, v kombinaciji z drugimi TCP/IP protokoli pa tudi za prenašanje IP datagramov, je dejansko sestavljen iz treh nivojev:

nivo 1 - fizični nivo (uporablja X.21, X.21 BIS ter RS232-C standard),
nivo 2 - podatkovni nivo (uporablja LABP podmnožico HDLC-ja),
nivo 3 - mrežni nivo, ki vsebuje DTE/DCE specifikacije povezav.

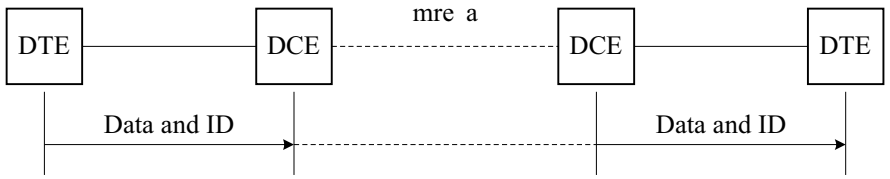
Najpogostejše storitve, ki jih protokol X.25 nudi so:

- uporaba virtualnih povezav, ki jih glede na trajanje povezave delimo na:
  - trajna virtualna vezja (PVC - Permanent Virtual Circuit), pri katerih so uporabniki na PSDN omrežje priključeni na podoben način kot pri najetih telefonskih vodih in
  - komutirana virtualna vezja (SVC - Switched Virtual Circuit), pri katerih PSDN omrežje vzpostavi virtualno povezavo med dvema uporabnikoma za čas trajanja komunikacije,
- spreminjanje propustnosti razredov in skupin, ki lahko variira od 7 bps do 48 kbps,
- hitra izbira,
- razširitev paketnega označevanja z uporabo sprejemnih in oddajnih števcev P(s) in P(r) z 8 na 128

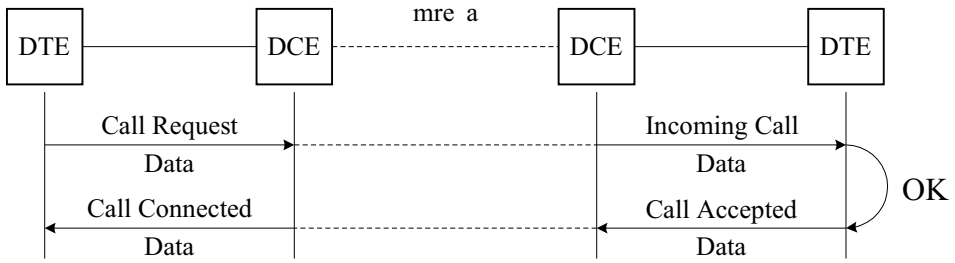
- ponovno pošiljanje paketov z uporabo P(s) oziroma P(r),
- uporaba enosmernih logičnih kanalov, ki omejujejo logične kanale na izhodne ali vhodne klice,
- uporaba zaprtih skupin uporabnikov,
- povratno zaračunavanje tarife klica in
- uporaba nestandardnih dolžin paketov.



Slika 3.14: X.25 komutirano virtualno vezje

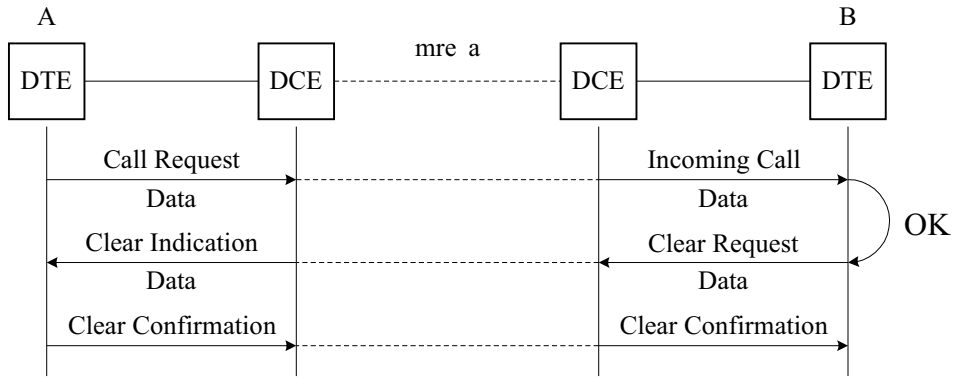


Slika 3.15: X.25 trajno virtualno vezje



Slika 3.16: X.25 hitri klic za izbiranje

Kadar hoče terminal A vzpostaviti zvezo s terminalom B s slike 3.14, sproži postopek klicanja, ki vključuje postopka "call request/incoming call", na kateri se klicani terminal B odzove s postopkoma



Slika 3.17: X.25 hitra izbira z vmesnim klicem

"call accepted/call connected", če sprejme zvezo s terminalom A. Postopek komunikacije med terminaloma se nadaljuje s prenosom podatkov in zaključuje s postopkoma "clear request/clear indication" in "clear confirmation". Druga možnost komunikacije med terinaloma, ki jo omogoča protokol X.25, je hitra izbira s slike 3.17, ki omogoča prenos podatkov kar znotraj samega klica. Po sprejemu klica in izvedbi postopka "call request/incomming call", ki v tem primeru vsebuje tudi podatke, pošlje terminal B skupaj s podatki za terminal A zahtevo po koncu prenosa "clear request". V tem primeru mora začetnik izmenjave podatkov, torej terminal A, potrditi konec izmenjave s postopkom "clear conformation", ki tokrat ne vsebuje podatkov.

Ideja hitrega povezovanja podpira aplikacije, kjer uporabniki med seboj izmenjujejo relativno majhne količine podatkov, kot so npr. trgovinski terminali, verifikacija kreditnih kartic, prenos bančnih sredstev itd.

Paketi, kot so seje, se med uporabniki DTE enote razpoznajo po številki logičnega kanala. Vsak paket vsebuje za ta namen polja za identifikacijo skupine kanalov (0 - 15) in posameznega kanala znotraj skupine (0 - 255). X.25 torej omogoča v vsakem fizičnem kanalu 4096 logičnih kanalov, od katerih vsak deluje povsem neodvisno. Število logičnih kanalov, po katerih se izvajajo virtualni klici, se v posameznem omrežju določi posebej v fazi določitve mrežne strukture.

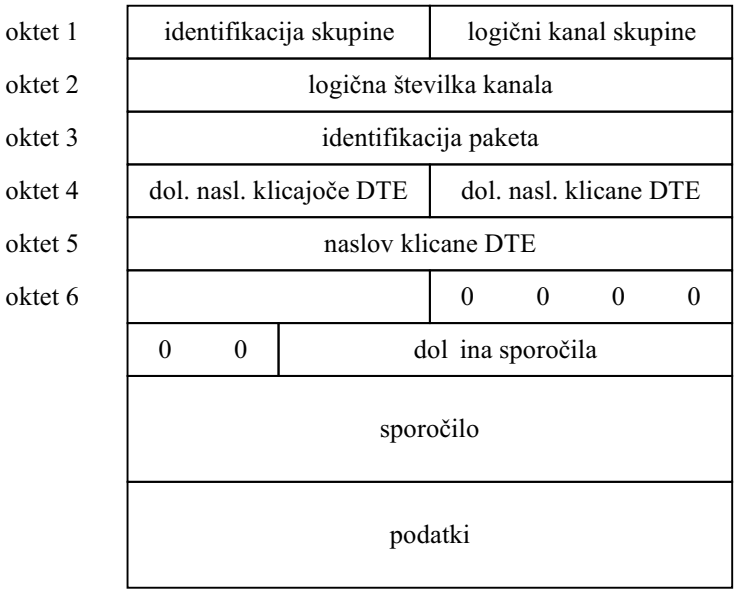
Logična številka kanala omogoča identifikacijo seje med dvema ali večimi uporabniki v paketnem omrežju z uporabo ene same transportne povezave.

Obliko paketa, ki se v X.25 protokolu uporablja za vzpostavitev zveze med terminaloma, prikazuje slika 3.18.

Polja začetka paketa so namenjena identifikaciji oblike paketa. Sledijo jim polja logične skupine kanala in številka logičnega kanala, ki opisujejo ustrezno pripadnost kanala. Tretja beseda paketa X.25 protokola identificira tip paketa (tabela 3.2), ki lahko predstavlja vzpostavitev ali prekinitev klica, podatkovne pretoke itd. Dolžini naslovov in naslovni polji vsebujeta informacije o klicočem in o klicanem DTE. Paketom je pogosto dodano tudi polje sporočil, kateremu sledi podatkovni del paketa, ki dejansko vsebuje tudi polja, katera omogočajo pošiljanje paketov med obema stranema. Polja, ki imajo dodane števce oddaje in sprejema določene podatkovne sekvence P(r) oziroma P(s), se nahajajo v tretjem oktetu naslovnega dela podatkovnega polja.

DTE → DCE	DCE → DTE	Biti tretjega okteta
call request	incoming call	0 0 0 0 1 0 1 1
call accepted	call connected	0 0 0 0 1 1 1 1
clear request	clear indication	0 0 0 1 0 0 1 1
clear confirmation	clear confirmation	0 0 0 1 0 1 1 1

Tabela 3.2: Tipi X.25 paketov



Slika 3.18: Oblika request/incoming call paketa

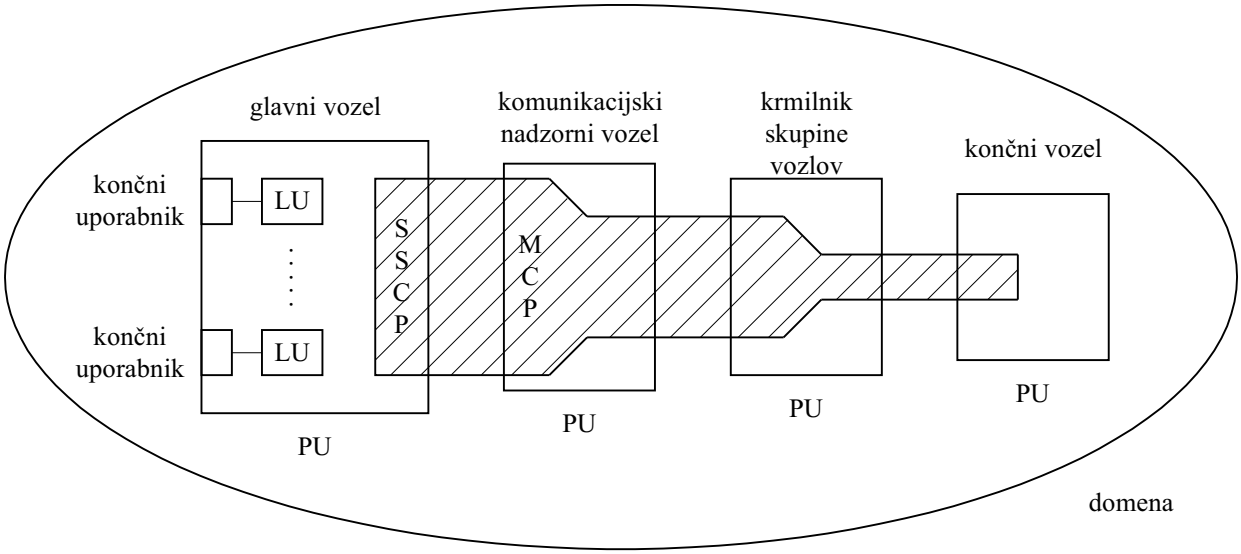
Potek nadzora pri X.25 je podoben poteku, ki smo ga podali v prejšnjih poglavjih. Nadzor je izveden dvosmerno za vsak logični kanal na vsaki postaji DTE z uporabo nadzornih polj oddaje in sprejema P(s) in P(r).

3.6 SNA model

Za razliko od predhodno obravnavanega ISO-OSI modela, je bil SNA (System Network Architecture) protokol razvit neposredno v računalniški industriji in je bil kot tak namenjen povezovanju IBM-ovih osrednjih računalnikov in terminalov. Večnivojski SNA protokol, danes predstavlja zaradi zmožnosti podpiranja istoležnih komunikacij nadgrajen v APPN (Advanced Peer-to-Peer Networking) protokol, enega uporabnejših in v praksi pogostejših protokolov za uporabo v omrežjih s porazdeljeno obdelavo podatkov.

SNA protokol je zasnovan na domenah, ki vključujejo (slika 3.19):

- glavni vozel, kateri vsebuje nadzorno servisno točko (SSCP - System Service Control Point), ki predstavlja osrednjo točko pri upravljanju konfiguracije mreže in katere glavne naloge v SNA omrežju so:
  - zasnova uporabnikovih sej v omrežju,
  - izvajanje posameznih operacij med elementi neke domene,
  - nadzor nad skupnimi viri domen,
  - vzpostavitev, vzdrževanje in zaključitev dela v omrežju,
  - odkrivanje in razvrščanje napak,
  - prikazovanje stanja omrežja in
  - koordinacija aktivnosti fizičnih in logičnih enot,
- komunikacijski nadzorni vozel (CUCN - Communucation Controller Node), ki predstavlja čelni procesor (FEP - Front End Procesor),



Slika 3.19: Vozli v domeni SNA omrežja

- krmilnik skupine vozlov (CCN - Cluster Controller Node), ki je dejansko periferna naprava in se uporablja za nadzor niza perifernih enot in
- končni vozel, ki predstavlja najbolj oddaljeni vozel v SNA porazdeljenem sistemu.

Poleg že omenjenih SSCP pa sestavljajo SNA omrežjih še fizične enote (PU - Physical Unit) in logične enote (LU - Logical Unit), ki vse spadajo med omrežne naslovljive enote (NAU - Network Addresable Unit).

Vsak vozel v SNA omrežju vsebuje fizično enoto, ki v nasprotju s svojim imenom, ne spada nujno med strojno opremo omrežja. Zelo pogosto je namreč posebno pri programabilnih osrednjih računalnikih in komunikacijskih kontrolerjih realizirana s programsko opremo. Fizična enota, ki deluje pod nadzorom SSCP, deluje kot vstopna točka v omrežje za eno ali več logičnih enot.

Logična enota deluje kot vstopna točka v SNA omrežje, tako da zagotavlja potrebne pretvorbe ter nadzor pretoka podatkov in vso programsko opremo, ki je potrebna končnemu uporabniku. Komunikacija med dvema končnima uporabnikoma, ki v SNA omrežjih ne komunicirajo s paketi ampak s sejami, potemtakem predstavlja komunikacijo med dvema logičnima enotama. Podobno kot pri paketnem prenosu, pa je tudi pri SNA možno povezovanje med več vozlišči na nekem področju; tej tehniki pravimo kar tehnika večkratnega povezovanja.

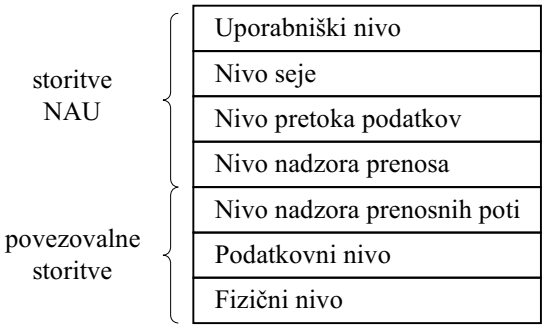
3.6.1 Funkcijski nivoji SNA modela

Podobno kot večina danes v praksi uporabljenih komunikacijskih protokolov, spada tudi SNA model protokolov med večnivojske protokole. Storitve, ki jih nudi SNA model, lahko združimo v dvoje osnovnih skupin:

- storitve NAU (Network Address Unit Service)
- povezovalne storitve (Path Control Network Service)

Uporabniški nivo SNA modela opravlja funkcije aplikacijskega in predstavitvenega nivoja ISO-OSI modela.

Nivo seje je razdeljen v tri delovne nivoje, ki omogočajo izmenjavo sporočil med upravljalci omrežja, nalaganje programov, naslavljanje in pretvorbo logičnih imen v ustrezne fizične naslove v mreži.



Slika 3.20: Funkcijski nivoji SNA modela

Nivo pretoka podatkov omogoča izvajanje določenih funkcij ISO-OSI transportnega nivoja. Tako podpira popoln dvosmerni (polni dupleks) in enosmerni (polovični dupleks) dialog med posameznimi NAU. V tem načinu lahko pošiljajoča NAU preusmeri dialog tako, da sprejemni enoti NAU zaukaže, ki potemtakem predstavlja le en člen v komunikacijski verigi, naj tudi ona prične z oddajanjem. Vsaka napaka v taki verigi povzroči, da se vsi delni prenosi med elementi v verigi zavržejo.

Sporočila, ki se prenašajo po SNA omrežju, se potrde takoj, ko jih sprejme LU. Nadzorni program za prenos sporočil omogoča potrjevanje tudi izključno ob določenih dogodkih, kot je delno potrjevanje sporočil, ali potrjevanje le v primeru nastopa napake itd.

Nivo nadzora prenosa ohranja stanje aktivne seje in povezuje podatke skozi vse nivoje do ustrezne NAU.

Ta nivo omogoča upravljanje z okni, tako da LU lahko nadzira prenos niza paketov, obdelavo sporočil in sprosti pomnilnik, namenjen shranjevanju sporočil. Nadzorni nivo omogoča uporabo glav, ki se pri prehodu skozi posamezne funkcijske nivoje SNA modela priključujejo k sporočilu.

Nivo nadzora prenosnih poti je odgovoren za povezovanje in nadzor pretoka. Ta del je tudi odgovoren za segmentacijo sporočil, saj se učinkovitost prenosa z drobljenjem sporočila v manjše enote različnih velikosti lahko močno poveča. Hkrati nivo nadzora prenosnih poti nadzira povezovalne poti kot bloke sporočil in s tem zmanjšuje število potrebnih V/I operacij.

Ta nivo vsebuje tudi takoimenovane virtualne poti povezav.

Podatkovni nivo in fizični nivo SNA modela sta zadolžena za opravljanje podobnih funkcij kot istoimenska nivoja ISO-OSI modela in sta najpogosteje realizirana s SDLC oziroma z RS232-C in X.21 fizičnim vmesnikom.

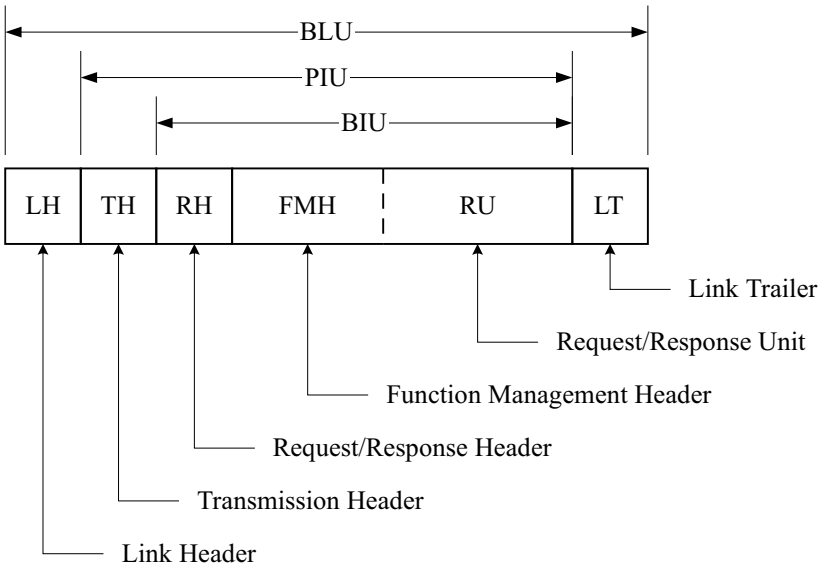
3.6.2 SNA sporočila in povezave

SNA sporočila

Format SNA sporočila, kateremu se pri prehodu skozi posamezne funkcijske nivoje SNA modela dodajajo pripadajoče glave, je prikazan na sliki 3.21.

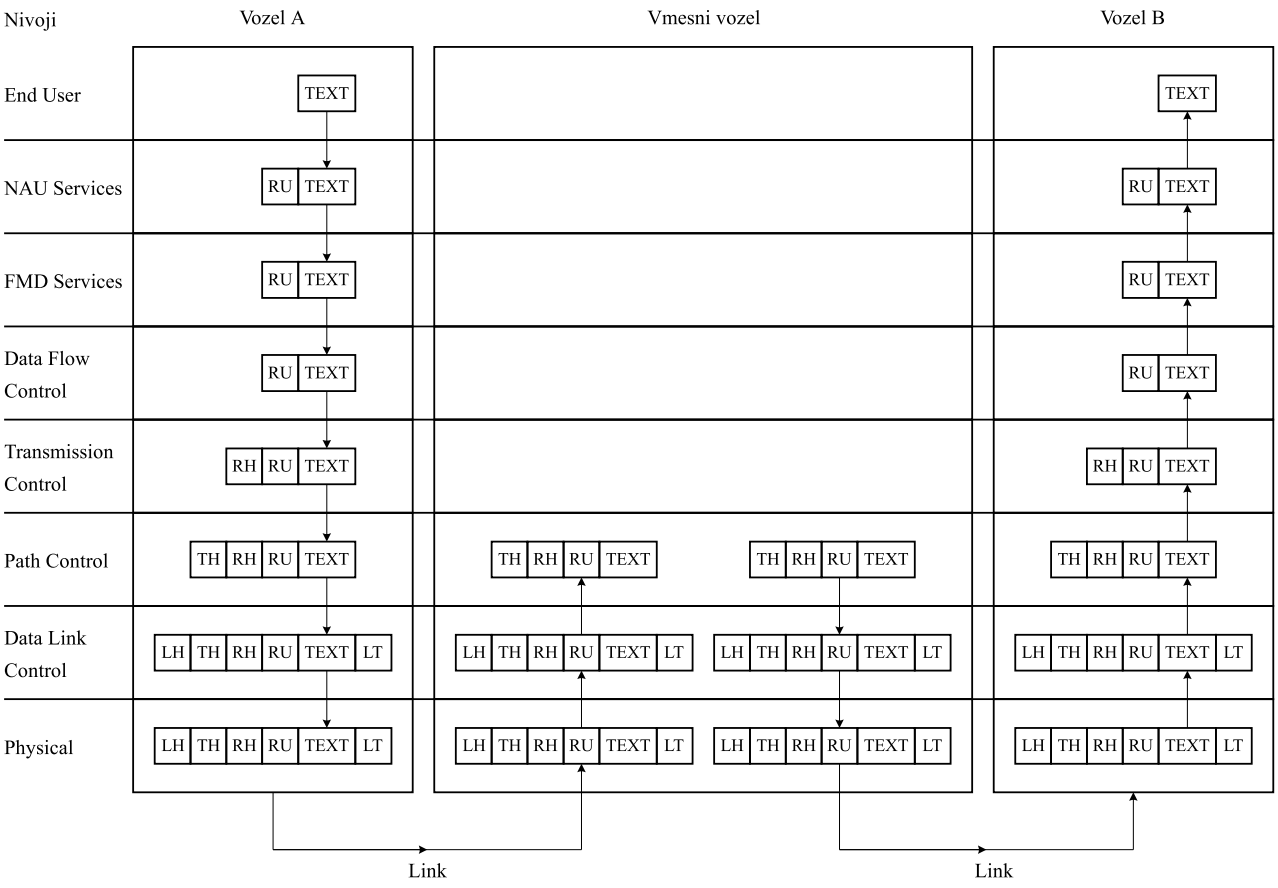
SNA povezave

Takoj ko uporabnik prične z uporabo SNA omrežja, mora določiti razred storitev (COS - Class Of Service), da bi upravljalec mreže lahko določil priporočljivo pot in nivo povezave. Razred storitev določa lista možnih povezav, ki jim pravimo tudi virtualne povezave. Virtualna povezava je dejansko logična pot med dvema končnima uporabnikoma, ki se v fazi prenosa preslika v tabelo dejanskih



Slika 3.21: SNA sporočilo

poti. Le-ta pa predstavlja niz vozlišč, kateri se pričnejo in končajo na nivoju uporabnikove LU in se pogosto še naprej drobe na segmentne poti. Tabele povezav vsebujejo tako naslove podpodročij končnih uporabnikov, kot tudi številke povezovalnih poti.



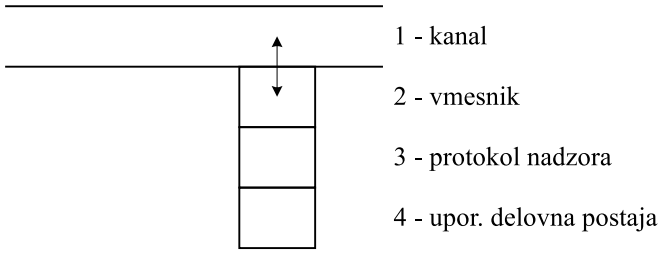
Slika 3.22: SNA povezave

3.7 Lokalne mreže

Lokalne mreže postajajo danes predvsem po zaslugi vse večje uporabnosti v poslovnih sistemih, kjer se že več kot 90% vseh informacij prenaša po lokalni mreži, vse bolj razširjene. Če so se v šestdesetih letih gradile mreže, ki so omogočale neposreden dostop manjših računalnikov, večinoma terminalov, do velikih poslovnih računalnikov in so se v sedemdesetih letih mreže uporabljale predvsem za dostop do skupnih virov, danes le-te predstavljajo osnovni povezovalni sistem vsake manjše enote. Tipične prenosne hitrosti lokalnih mrež se danes v odvisnosti od tehnoloških izvedb gibljejo med 10 in 100 Mbps.

3.7.1 Komponente lokalnih mrež

Lokalno mrežo sestavljajo komponente iz slike 3.23. Najnižji nivo lokalne mreže predstavlja prenosni medij, med katerimi se danes uporabljajo koaksialni kabli ali optični vodniki. Pogosto se v praksi uporabljajo tudi prepletene telefonske palice, dandanes pa se predvsem zaradi povečanja prenosne hitrosti (nad 300 Mbps na razdalji do 14 km) in zmanjšanja izgub uporabljajo optična vlakna. V zadnjem času se prav tako pogosto predvsem v prenosnih aplikacijah pojavljajo infrardeči prenosni sistemi, ki omogočajo prenosne hitrosti do 100 kbps na razdalji do 1 km.



Slika 3.23: Komponente lokalne mreže

Na drugem nivoju se pojavlja več tehnoloških rešitev vmesnikov med prenosnim medijem in nadzornim protokolom; od enostavnih konektorjev za koaksialni kabel, infrardečih in laserskih diod do sprejemnikov steklenih vlaken itd. Nekatera omrežja imajo na tem nivoju vgrajene ojačevalne elemente, spet druga nekatera pa le-te vsebujejo na nivoju priključka terminala (RS232-C).

V tretjem nivoju komunikacijski protokoli nadzirajo pretok podatkov po mreži in omogočajo uporabniku dostop do mreže. Večina protokolov na tem nivoju deluje po principu opisanem v prejšnjih poglavjih.

Zadnji, četrti, nivo zavzemajo terminali in večji računalniški sistemi.

3.7.2 Mrežni protokoli

V letu 1983 je bil za protokole lokalnih mrež izdan IEEE 802 standard, katerega najpomembnejši sestavni deli so:

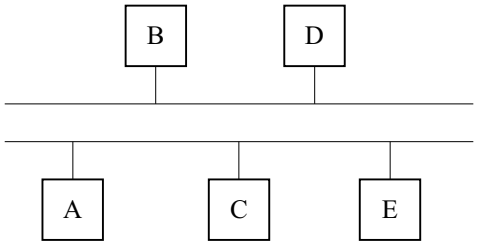
- 802.2 - LLC (Logical Link Control),
- 802.3 - CSMA/CD (Carrier Sense Multiple Access/Collision Detect),
- 802.4 - Token Bus in
- 802.5 - Token Ring.

Evropski proizvajalci programske in strojne opreme ECMA (European Computer Manufacturers Association) so za svoj standard sprejeli standard IEEE 802.5 pod oznako ECMA 89, medtem ko je ISO sprejel standard IEEE 802 pod oznako ISO 8802.

LLC (IEEE 802.2) omogoča izvedbo mrežnega vmesnika standardov 802.3, 802.4 in 802.5. LLC protokol, ki je podmnožica HDLC protokola in uporablja uravnotežen način prenosa, omogoča generiranje in interpretiranje ukazov nadzora nad pretokom podatkov in določa ravnanje oddajnih in sprejemnih postaj v primeru detekcije napake pri prenosu.

CSMA/CD (IEEE 802.3) protokol s kolizijo so razvili na osnovi ALOHA mreže na University of Hawaii. Raziskovalna mreža ALOHA je bila zasnovana na radijskem prenosnem sistemu, v katerem so sekundarne postaje neodvisno druga od druge oddajale signale. Glavna postaja te mreže je oddajala na enem frekvenčnem pasu, ostale postaje pa na drugem frekvenčnem pasu. Zaradi naključnosti prenosa sekundarnih postaj, je med njimi pogosto prihajalo do kolizije. Ker so po vsaki koliziji postaje nekaj časa počakale z oddajanjem, je tak pristop omogočal le 18% izkoriščenosti prenosnega kanala. Modificiran sistem ALOHA, pri katerem se je vsaka sekundarna postaja najprej sinhronizirala z urinim signalom glavne postaje in pričela oddajati le v določenih intervalih, pa je omogočal 36.8% izkoriščenost kanala.

CSMA/CD protokol uporablja nekaj zgoraj omenjenih konceptov sistema ALOHA. Pred oddajanjem postaja vedno posluša signal na prenosni poti in z oddajanjem ne prične vse dokler je signal na njej prisoten. Ko postaja na prenosni poti ne zazna več signala, odda svoje sporočilo. Seveda CSMA/CD protokol zaradi možnosti kolizije vseskozi opazuje dogodke na prenosnem kanalu v času oddajanja sekundarne postaje in v primeru kolizije o tem obvesti vse postaje v sistemu. Ker pa CSMA/CD protokol ne pozna glavne postaje, kot jo pozna mreža ALOHA, le-ta ne uporablja dodatnega kanala, kjer glavna postaja razpošilja BUSY signal vsem sekundarnim postajam.



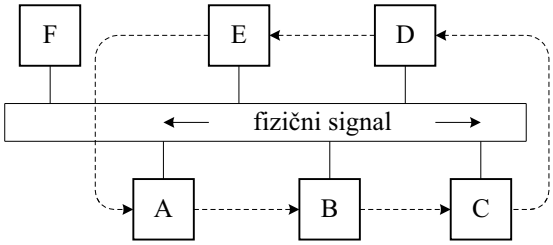
Slika 3.24: CSMA/CD topologija s skupnim vodilom (IEEE 802.3)

Pri CSMA/CD protokolu vsaka sprejemna postaja zazna prispeli okvir, ki pripada sinhronizaciji vhodnega signal. Sprejeti signal se nato dekodira in prevede v ustrezni binarni podatkovni niz. Sprejemnik se na podlagi naslovnega dela okvirja in vrstnega reda prenosa okvirjev ter morebitnih napak pri prenosu odloči, ali bo ta okvir sprejel ali ne. Če ga sprejme, le-ta prenese podatke uporabniku.

CSMA/CD omrežja odlično delujejo v sistemih z večtočkovno topologijo in asinhronim prenosom, kjer so vse postaje povezane s skupnim vodilom.

Token Bus (IEEE 802.4) komunikacijski protokol uporablja za dostop do vodila posebno podatkovno strukturo, imenovano žeton, ki v logični krožni mreži prehaja iz postaje na postajo in je lahko predstavljen z ustrezno časovno rezino ali okvirjem. Čim postaja sprejme nanjo naslovljen žeton, prične s prenosom podatkovnih paketov, saj ima takrat ekskluzivni dostop do vodila, ali pa ga preusmeri k določeni postaji z zahtevo po medsebojnem prenosu podatkov. Takoj ko trenutno aktivna postaja zaključi z oddajanjem, pošlje žeton svojemu nasledniku v logični zanki. V primeru, da je žeton naslovljen na trenutno izključeno postajo, bo enota, ki je sprožila proces prenosa podatkov, po izteku njej namenjenega čas žeton predala naslednji znani postaji zanke.

S slike 3.25 je razviden sekvenčni dostop postaj do vodila. Postaja, ki kot postaja F iz slike 3.25 ni del logične zanke, sicer lahko sprejema podatke in se nanje odziva, ne more pa generirati



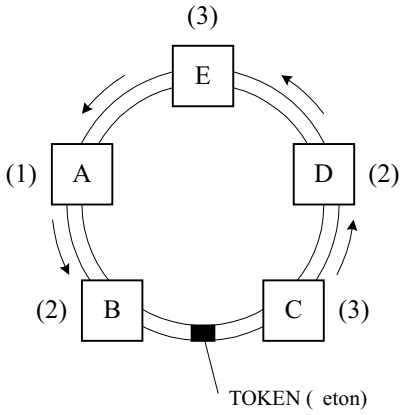
Slika 3.25: Vodilo z žetonom in logična zanka (IEEE 802.4)

žetona in tako sama odločati o dodeljevanju dostopa do skupnega vodila. Postaje se logičnemu vodilu IEEE 802.4 dodajajo s postopkom imenovanim odziv na klic:

- Takoj ko postaja sprejme žeton, sproži postopek pridobivanja svojega naslednika, ki mu bo po izteku njej določenega časa, navadno je enak dvojnemu času prenosa žetona med najbolj oddaljenima točkama vodila, predala žeton.
- V primeru, da na njen postopek pridobivanja naslednika ni odziva, se žeton prenese na naslednjo znano postajo.
- V primeru, da prejme odziv nove in njej neznane postaje, le-ta pošlje žeton na naslov, ki ga je sprejela v odzivu.
- Postaja, ki je ravnokar vstopila v mrežo, sprejme žeton, postavi svoje naslove in nadaljuje s postopkom.

Seveda pa se postaja lahko tudi izloči iz logičnega vodila. Takoj ko tovrstna postaja sprejme žeton, ga pošlje svojemu prehodniku, ki nato omogoči prenos žetona na njej sledečo postajo.

Token ring (IEEE 802.5) protokol je prav tako kot protokol IEEE 802.4 protokol z uporabo žetona.



Slika 3.26: Obroč z žetonom (IEEE 802.5)

Podobno kot pri zgoraj opisanem IEEE 802.4 protokolu tudi pri protokolu IEEE 802.5 zaradi preprečevanja kolizije žeton kroži po mreži, ki pa v tem primeru sovpada s fizično zanko. V vsakem vozlišču se torej primerja naslov sprejemnika z naslovom v glavi žetona in v primeru, da le-ta sovpadata, se žeton zadrži in prebere. Sprejemna postaja sporoči sprejem podatkov pošiljatelju s postavitvijo zastavic stanja v samem žetonu, tako da lahko vozel ali postaja, ki je ustrezni žeton poslala, sedaj izprazni vsebino žetona in ga prostega pošlje nazaj v obroč. V IEEE 802.5 ni dopustna uporaba večih žetonov hkrati, kot tudi ne večkratna uporaba istega žetona s

strani iste postaje. Je pa v IEEE 802.5 protokolu dovoljena uporaba prioritet, ki se določajo z nastavitvijo ustreznih indikatorjev prioritete v samem žetonu.

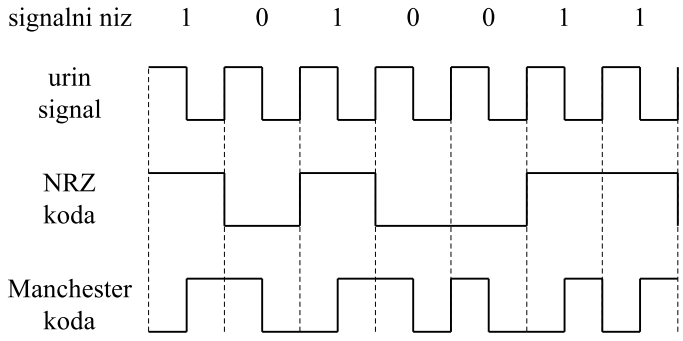
Proces uporabe žetona v krožni mreži je prikazan sliki 3.26. Vzemimo, da ima postaja A najnižjo prioriteto (1), postaji C in E pa najvišji prioriteti (3). Postaje postavljajo v posebno polje rezervacij samega žetona svoje prioritete, ki določajo vrstni red dostopa postaj do vodila v naslednjem obhodu žetona. V primeru, da morajo vse postaje oddati podatke, se bo žeton najprej ustavil na postaji C, ki ima najvišjo prioriteto, in nato v naslednjem obhodu na postaji E, ki ima sicer enako prioriteto kot postaja C, vendar pa je na poti žetona po omrežju za njo. Seveda pa se v praksi postaje ne postavljajo vseh prioritet v vsakem obhodu žetona, saj običajno postaje z največjo prioriteto ne oddajajo podatkov tako pogosto.

3.7.3 Praktični primeri lokalnih mrež

Ker je trenutno na tržišču na voljo cel niz med seboj arhitekturno zelo različnih mrež, si v nadaljevanju oglejmo le nekaj najbolj tipičnih.

Ethernet

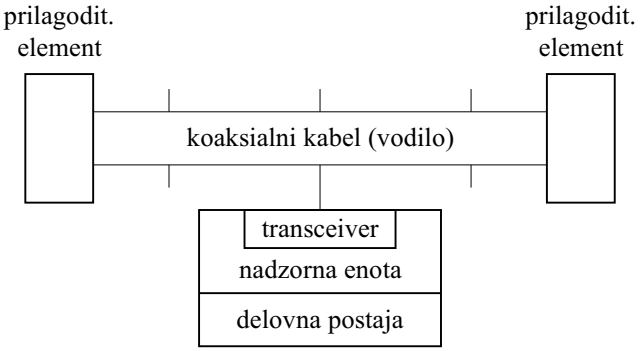
Ethernet omrežja, ki so danes daleč najbolj razširjena, so bila razvita pri družbi Xerox v sedemdesetih letih, leta 1980 pa sta DEC in Intel objavila skupne specifikacije mreže, zasnovane na konceptu Etherneta. Ethernet omrežja uporabljajo IEEE 802.3 CSMA/CD protokol in signalizacijo na osnovnem pasu (baseband signalling), za povezavo pozameznih mrežnih komponent v lokalno omrežje pa uporablja oklopljen koaksialni 50 Ω kabel največje dolžine 500 m, ki omogoča prenosno hitrost 100 Mbps z možnostjo vključevanja do 1024 postaj v lokalno omrežje. Ethernet omrežja uporabljajo za kodiranje binarnih NRZ signalov pri prenosu Manchester kodo, pri kateri logični ničli NRZ niza ustreza na sredini urine periode prehod iz visokega v nizki nivo, logični enici pa prehod iz nizkega v visoki napetostni nivo. Tako kodiranje ima pred običajnim NRZ kodiranjem vrsto prednosti, saj zaradi prehoda med logičnima nivojema, ki se vsakič dogodi na sredini urine periode, v sistemih, ki uporabljajo Manchester kodiranje, navadno odpade potreba po sinhronizaciji med oddajnikom in sprejemnikom. Ker pa je povprečna vrednost signala, kodiranega z Manchester kodo, v vsakem trenutku enaka nič, so enosmerni tokovi, ki nastanejo kot posledica sprememb lokalne koncentracije naboja, bistveno manjši, razdalje med regeneratorji v omrežju pa posledično večje.



Slika 3.27: NRZ in Manchester koda

Tipično Ethernet omrežje s slike 3.28 sestavljajo koaksialni kabel, prilagoditveni elementi, linijski ojačevalniki, nadzorne enote in delovne postaje.

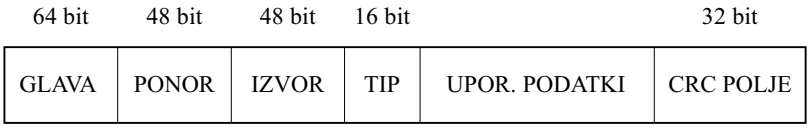
Prilagoditveni elementi zaključujejo linijo in preprečujejo elektromagnetno sevanje nezaključenih koncev linij v okolico. Linijski ojačevalniki oddajajo in sprejemajo signale, zaznajo kolizijo paketov in vzdržujejo ustrezno kvaliteto signalov na samem vodilu. Nadzorne enote omogočajo upravljanje



Slika 3.28: Arhitektura Ethernet omrežja

kolizij, kodirajo in dekodirajo signale in opravljajo druge naloge v okviru CSMA/CD. Delovne postaje na nivoju končnih uporabnikov omogočajo obdelavo tekstov in baz podatkov, programiranje itd.

Ethernet paket, ki ga prikazuje slika 3.29, je sestavljen iz šestih polj.



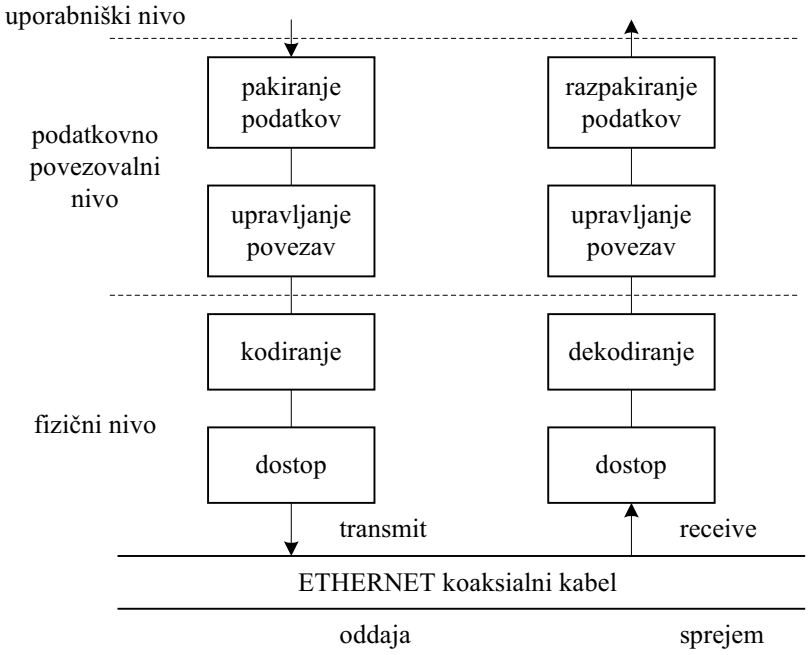
Slika 3.29: Ethernet paket

Glava ali uvod paketa je uporabljena za stabilizacijo kanala in sinhronizacijo med oddajno in sprejemno postajo. Dolžina glave, ki sestavljajo ničle in enice in se vedno konča z dvema zaporednima enicam, je 64 bitov. Naslova izvora in ponora, oba dolžine 48 bitov, identificirata oddajno in sprejemno enoto. Naslov ponora je lahko tudi skupinski ali celo globalni naslov. Polje tipa podatkovnega paketa dolžine 16 bitov je namenjeno končnemu uporabniku in se uporablja za enoličen dogovor med višjimi nivoji. Polje uporabnikovih podatkov, ki je namenjeno prenosu podatkov med uporabnikoma, vsebuje najmanj 46 oktetov podatkov, tako da je najmanjša dolžina Ethernet paketa enaka 64 oktetov, in največ 1.500 oktetov. Polje CRC (Cyclic Redundancy Check) dolžine 32 bitov je namenjeno verifikaciji pravilnosti prenosa podatkov. Njegova vrednost je odvisna od vsebine obeh naslovnih polj, tipa podatkovnega paketa in uporabnikovih podatkov in se izračuna tako na oddajni kot tudi na sprejemni strani.

Ethernet protokol je klasični primer večnivojskega komunikacijskega protokola, saj je vpet med tri nivoje:

- uporabniški nivo, ki predstavlja nivo delovne postaje,
- podatkovni povezovalni nivo, ki od uporabniškega nivoja sprejema podatke in jih povezuje v pakete ter upravlja s povezavami in
- fizični nivo, ki omogoča kodiranje podatkov in dostop do prenosnega kanala.

Prenos paketa Z uporabniškega nivoja podatki preidejo na podatkovno povezovalni nivo, kjer se najprej združijo v pakete in nato preidejo na nivo upravljanja povezave. Podatkovno-povezovalni nivo hkrati opazuje tudi dogajanje na vodilu in takoj ko je vodilo prosto, pošlje predelani niz bitov na fizični nivo. Nivo kodiranja takoj pošlje na vodilo že omenjeno glavo podatkovnega paketa, ki omogoči vsem prejemnikom in ojačevalnikom, da se sinhronizirajo na uro oddajne enote in opravi pretvorbo binarne NRZ kode v Manchester kodo. Nivo za dostop na kanal generira ustrezne signale na koaksialnem kablu in opazuje dogajanje na vodilu.



Slika 3.30: Nivoji Ethernet protokola

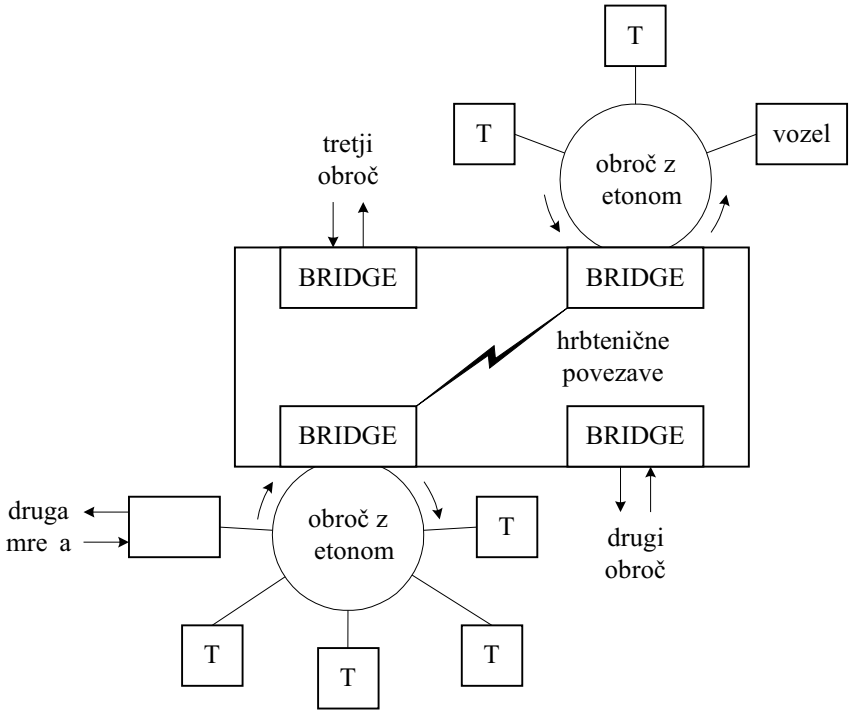
Sprejem paketa Enota za detekcijo v fizičnem nivoju sprejemnika zazna prihajajoč signal. Podatkovno-povezovalni nivo se sinhronizira s prihajajočo glavo paketa, dekodirno vezje dekodira kodiran signal v binarni NRZ niz, odvrže uvodne sinhronizacijske bite in pošlje spakirane podatke na višji nivo za razpakiranje podatkov. Paket upravljanja povezav detektira nosilec signala, ki ga dobi z najnižjega nivoja ojačevalnikov in testira naslov prejemnika. Če je le-ta pravilen, pošlje podatkovni okvir na nivo uporabnika. Na tem nivoju se tudi verificira pravilnost prejetih podatkov in v primeru okvare, pošlje informacijo o napaki uporabnikovi enoti.

Upravljanje kolizije Signal neke postaje lahko sovpada s signalom druge postaje. Če npr. signal postaje A ne pride do postaje B preden le-ta prične z oddajanjem lastnega paketa, tedaj signala sovpadata. Perioda ranljivosti se imenuje okno kolizije in je določena s časom širjenja vala po celotnem kanalu. Prekrivanje dveh signalov na kanalu zazna nivo sprejemnika/oddajnika in v primeru kolizije pošlje signal ustreznemu nivoju upravljanja povezave, ki oddajni postaji takoj odpošlje bitni niz imenovan "jam" (blokada) in jo s tem informira o napaki pri prenosu. Nivo upravljanja povezave zato zaključi s prenosom in z naključno zakasnitvijo prične s ponovno oddajo. Sprejemna postaja pa pri vsem tem ne zazna kolizije signala, temveč razbitje okvirja, kar je dovolj za zavrnitev sprejetega paketa.

IBM Token Ring

Leta 1985 je IBM naznanil svoj lastni mrežni protokol znan pod imenom IBM Token Ring. IBM-ova topologija dopušča povezavo več obročev preko hitrih vmesnikov. Ti vmesniki - mostovi - so povezani na hrbtenični osnovni obroč, ki omogoča križne mrežne povezave s kopiranjem okvirjev iz enega obroča na drugega. Mostovi omogočajo tudi prehod iz ene prenosne hitrosti na enem obroču na drugo hitrost prenosa na drugem obroču. V vsakem primeru pa vsak obroč, povezan preko hrbteničnih povezav z drugimi obroči, ohranja svojo kapaciteto prenosa in lahko nemoteno deluje čeprav je na nekem drugem obroču prišlo do napake.

Hrbtenično povezavo tvorijo prepleteni kabli, po katerih poteka prenos z 10 Mbps ali optični vodniki s 100 Mbps prenosom. Hrbtenična povezava pa lahko deluje tudi kot osnovni hrbtenični obroč na



Slika 3.31: IBM-ov obroč z žetonom

širokopasovni povezavi, ki omogoča hitre povezave z drugimi sistemi ali celo z video sistemi (multimedijski sistem, video na poziv).

IBM-ov obroč z žetoni najpogosteje tvorijo prepletene telefonske parice ali optični vodniki. Obroč z žetonom uporablja kombinacijo zvezdaste topologije in topologije obroča. Fizično omogoča enosmerne povezave točka-točka po IEEE 802.5 standardu in povezavo do 250 delovnih postaj.

Vsaka postaja, ki je povezana na obroč, uporablja posebni priključek, ki izvaja osnovne funkcije fizičnega vmesnika in linijskega protokola. Ta vmesnik razpozna vse okvirje in žetone, omogoča detekcijo napak, dekodiranje naslovov itd. Pri uporabi prepletenega kabla je največja dolžina obroča, ki uporablja te vrste priključke, enaka 45 m.

Namesto SDLC protokola je na povezovalnem nivoju IBM uporabil IEEE 802.2 LLC protokol, saj v primeru uporabe SDLC protokola, omrežje ne bi podpiralo neuravnoteženega asinhronega prenosa in istoležnih (peer-to-peer) komunikacij na nivoju kanala.

GM MAP in TOP protokol

GM je močno vplival na razvoj mrež s svojim MAP (Manufacturing Automation Protocol) protokolom. Predstavlja enega prvih standardov razvitih neposredno v industriji in omogoča povezovanje raznih industrijskih strojev v enoten sistem. MAP protokol uporablja IEEE 802.4 vodilo z žetonom, na nivoju povezav pa uporablja IEEE 802.2 logični nadzor nad povezavo.

Zelo podoben temu protokolu je TOP (Technical and Office Products) protokol, razvit pri Boeing Computer Services. Glavna razlika med obema protokoloma je na vmesniškem nivoju, kjer TOP uporablja IEEE 802.3 protokol.

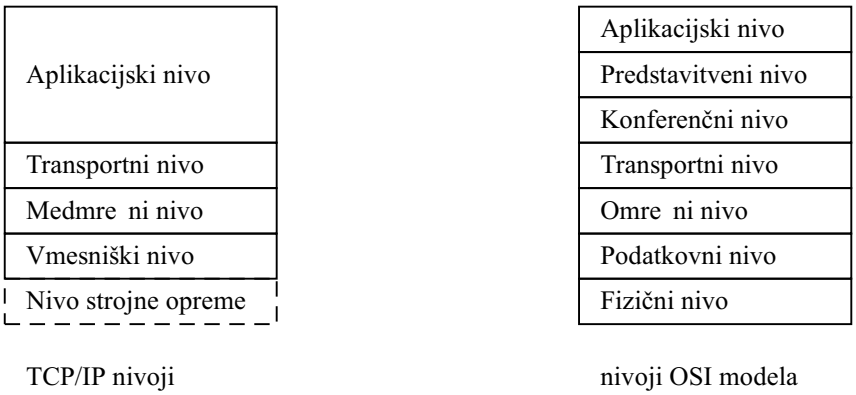
3.8 TCP/IP model

Družina TCP/IP protokolov je bila v sedemdesetih letih razvita za mrežo ameriškega obrambnega ministrstva ARPAnet (Advanced Research Projects Agency) z namenom povezovanja najrazličnejših

tipov računalnikov in ostale mrežne opreme v skupno omrežje. Danes je ta družina protokolov ena najbolj razširjenih, uporablja pa jo tudi Internet - izjemno raznolika zbirka računalniških mrež, od nekdanjega ARPAnet, NSFnet (National Science Foundation), do različnih vojaških in univerzitetnih mrež, in v zadnjem času vse večjega števila komercialnih uporabnikov.

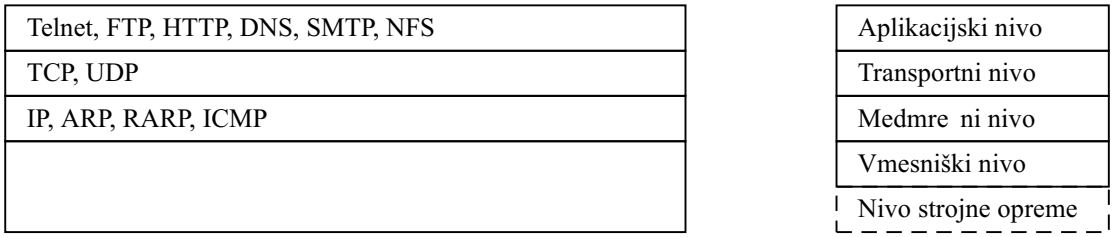
3.8.1 Funkcijski nivoji TCP/IP (Transmission Control Protocol/Internet Protocol) modela

Družino TCP/IP protokolov sestavljajo štirje funkcijski nivoji, ki so prikazani na sliki 3.32. Za primerjavo so prikazani tudi funkcijski nivoji ISO-OSI modela.



Slika 3.32: Funkcijski nivoji TCP/IP in ISO-OSI modela

Najpogosteje uporabljani protokoli posameznih nivojev TCP/IP modela pa so prikazani na sliki 3.33.



Slika 3.33: Najpogostejši protokoli TCP/IP modela

Aplikacijski nivo sestavljajo uporabniški programi, ki uporabljajo za komunikacijo računalniške mreže. Del programske opreme, ki se nahaja v tem nivoju, zagotavlja storitve direktnim uporabnikom terminalov in delovnih postaj.

V tem nivoju TCP/IP protokolov se nahaja cela množica protokolov, ki nudijo storitve uporabnikom, med katerimi so najpomembnejše:

- priključevanje oddaljenih uporabnikov v sistem (telnet)
- prenašanje datotek (FTP - File Transfer Protocol),
- uporaba WWW storitev (HTTP - HyperText Transfer Protocol)
- pošiljanje in sprejemanje elektronske pošte (SMTP - Simple Mail Transfer Protocol),
- uporaba mrežnega datotečnega sistema (NFS - Network File System), ki omogoča enostaven dostop do datotek na vseh računalnikih v omrežju

Transportni nivo V tem nivoju se nahajajo protokoli, ki pripravljajo sporočila za prenos po mreži. Najpomembnejša protokola, ki delujeta v tem nivoju sta protokol s spojem TCP (Transmission Control Protocol) in protokol brez spoja UDP (User Datagram Protocol). Uporaba UDP protokola ima v primerjavi s TCP protokolom prednost v aplikacijah, kjer vrstni red sprejetih datagramov in zanesljivost prenosa nista na prvem mestu.

Ker je večina računalniških sistemov, na katerih teče TCP/IP programska oprema, sposobna izvajati več uporabniških procesov istočasno, se kaj lahko zgodi, da hoče več procesov v gostujočem računalniku uporabiti iste TCP/IP komunikacijske protokole. Le-ti imajo tako za preprečevanje nastanka kritičnih situacij svoja vrata (protocol ports), ki so seveda različno naslovljena in tudi predpisana za večino standardnih aplikacij. Ko hoče neki proces poslati podatke prejmniškemu procesu, mora transportnemu protokolu sporočiti tri parametre: podatke, ki jih želi poslati, naslov prejemniškega računalnika ter naslov vrat na prejemniškemu računalniku, ki so naslovljena na proces, kateremu je sporočilo namenjeno.

Medmrežni nivo zagotavlja usmerjanje in druge prenosne funkcije, ki so potrebne, da se podatki prenesejo od pošiljatelja v eni mreži do prejemnika v isti ali drugi mreži. Medmrežni nivo deluje v pošiljateljskem in prejemniškem računalniku in v vseh usmerjevalnikih na poti med računalnikoma. V tem nivoju se vršijo odločitve, ki določajo katera pot med obema mrežama ali računalnikoma je najprimernejša. Pomembnejši protokoli, ki delujejo v tem nivoju so: IP (Internet Protocol), ICMP (Internet Control Message Protocol), ARP (Adress Resolution Protocol), RARP (Reverse Address Resolution Protocol) ter usmerjevalni protokoli.

IP protokol brez spoja je osnovni protokol skupine TCP/IP protokolov in se uporablja za prenos datagramov od pošiljatelja do prejemnika skozi mrežo.

ICMP je protokol, ki izpopolnjuje delovanje IP in omogoča gostiteljskemu računalniku, da opozarja na okoliščine v katerih prihaja do napak.

Vsak gostiteljski računalnik uporablja ARP in RARP protokola za vzdrževanje ARP in RARP skladišč, v katerih so shranjeni podatki, ki povezujejo 32-bitne IP naslove z navadno 48-bitnimi MAC (Medium Access Control) naslovi strojne opreme za uporabnike iste lokalne mreže.

Usmerjevalni protokoli so protokoli, ki jih najpogosteje poganjajo usmerjevalniki in s pomočjo katerih si le-ti izmenjujejo informacije, ki v vseh podrobnostih določajo pot posameznega paketa med pošiljateljem in sprejemnikom.

Vmesniški nivo opravlja povezavo s strojno opremo in predstavlja standardni vmesnik k medmrežnemu nivoju. Vmesniški nivo je odgovoren za sprejemanje sporočil iz medmrežnega nivoja in njihovo preoblikovanje v obliko, primerno za prenos preko mrežnih povezav kakršnekoli tehnologije. Ravno fleksibilnost protokolov tega nivoja je v veliki meri prispevala k praktični prevladi TCP/IP modela nad ISO-OSI modelom.

Nivo strojne opreme ni del TCP/IP protokolov in obsega vso fizično opremo potrebno za nemoteno delovanje komunikacijskega omrežja: omrežne vmesniške kartice, regeneratorje, mostove, usmerjevalnike, zvezdišča, priključke in kable.

Če hočemo torej zgraditi mrežo, ki bo delovala s pomočjo TCP/IP protokolov, lahko zanjo uporabimo kakršnokoli strojno opremo in kakršenkoli prenosni medij (npr. koaksialne ali optične vodnike). Vse kar potrebujemo za zagon takšne mreže, je primerna programska oprema, ki deluje v vseh zgoraj opisanih nivojih in omogoča uporabniškim aplikacijam dostop do fizične mreže.

3.8.2 Datagram in glava TCP in IP datagrama

Datagram

Povedali smo že, da se informacije v omrežjih, ki jih poganjajo TCP/IP protokoli, prenašajo kot zaporedja datagramov. Datagram je potemtakem zbirka podatkov, ki se obravnavajo in pošiljajo kot enotno sporočilo. Vsak datagram se pošilja skozi mrežo povsem neodvisno od ostalih in jih torej mreža obravnava popolnoma ločeno. Recimo da hočemo poslati 15.000 oktetov dolgo datoteko. Ker velika večina mrež iz čisto tehničnih razlogov tako dolge datoteke ne more prenašati, jo morajo protokoli transportnega nivoja razdeliti v približno 30 500-oktetnih datagramov. Vsak od teh datagramov bo poslan prejemniku, kjer jih bo le-ta sestavil v 15.000 oktetov dolgo datoteko. Dokler pa se datagrami prenašajo po mreži, prejemnik ne ve za način povezanosti. Povsem možno je torej, da bo datagram z zaporedno številko 13 prišel do prejemnika prej kot datagram z zaporedno številko 12. Prav tako se lahko zgodi, da bo nekje v mreži prišlo do napake in kakšen datagram ne bo prispel do prejemnika. V tem primeru bodo protokoli transportnega nivoja sprejemnika zahtevali njegovo ponovno oddajo.

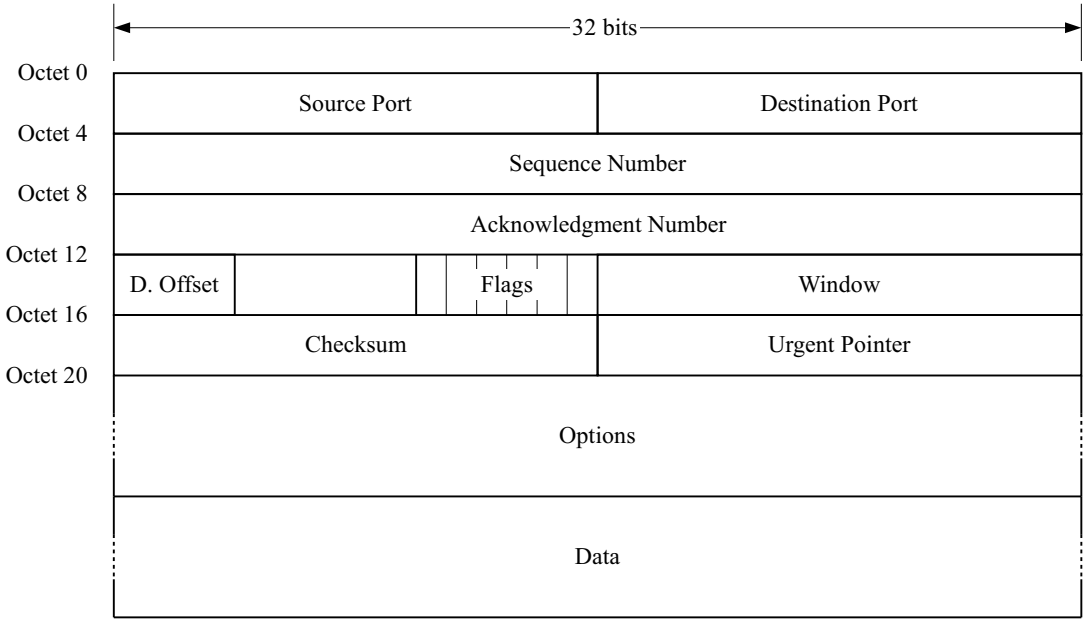
Pogosto se zdi, da sta izraza datagram in paket ekvivalentna, vendar je pri opisu TCP/IP protokolov izraz datagram pravilnejši. Datagram je namreč osnovna enota podatkov s katero se pri prenosu ukvarjajo protokoli, medtem ko je paket moč fizično detektirani izključno v prenosnem mediju med pošiljateljem in sprejemnikom. V večini primerov en paket vsebuje en datagram, lahko pa se zgodi, da se en datagram razdeli na več paketov (če se na primer TCP/IP datagrami prenašajo preko X.25 omrežja).

Seveda pa za uspešen prenos podatkov po omrežjih ni dovolj, da oddani datagram dospe na pravi naslov. TCP protokol sprejemnika mora namreč vedeti, kako sprejete datagrame povezati v enotno sporočilo in ugotoviti ali kakšen manjka. Pri tem se protokoli TCP/IP modela poslužujejo uporabe glav, ki jih na začetek datagrama postavijo zato, da ta datagram lahko zasledujejo. Čeprav pri pošiljanju in sprejemanju datagramov v veliki večini praktičnih primerov glavi, ki ju datagramu dodata TCP in IP protokola, zadostujeta za nemoten prenos podatkov med dvema mrežnima komponentama, pa se posebno pri prenosu podatkov med arhitekturno zelo različnimi omrežji, kjer je zaradi kompatibilnosti nujna uporaba protokolnih prevajalnikov (gateway), kaj lahko zgodi, da protokoli na datagram dodajo več glav.

Glava datagrama TCP

Recimo, da želimo prenesti neko datoteko iz enega računalnika na drug računalnik. Ker je sama datoteka predolga za enovit prenos, jo mora TCP, ob poznavanju največje dolžine, razdeliti v kose, ki jih lahko obdeluje v sodelovanju z ostalimi protokoli. Vsakemu takemu kosu datoteke TCP doda svojo glavo. Najpomembnejši njeni deli so zaporedna številka datagrama (Sequence Number) in številka vrat pošiljatelja (Source Port Number) ter številka vrat prejemnika (Destination Port Number), saj je v večini današnjih računalnikov, ki podpirajo hkratno izvajanje večjega števila procesov, vsakemu procesu enolično prirejena številka vrat. Če računalnik A pošilja datagrame računalniku B preko mreže, bo TCP računalnika A, določil številko vrat pošiljatelja in jo vpisal v glavo datagrama na ustrezno mesto. Prav tako bo TCP računalnika A ugotovil številko vrat, ki bo uporabljena na računalniku B (to številko določi TCP B), nakar to številko spet vpisal v glavo datagrama na ustrezno mesto. Če stran B pošilja datagrame strani A se številki vrat pošiljatelja in prejemnika seveda zamenjata. Glava vsakega datagram vsebuje tudi zaporedno številko datagrama, s pomočjo katere TCP na prejemniški strani iz datagramov v pravem vrstnem redu sestavi datoteko. Ker pa TCP ne šteje datagramov temveč oktete, ima prvi datagram pri 500-oktetnem datagramu številko 0, drugi 500, tretji 1.000 itn. Pomembno število v glavi TCP datagrama je tudi vsota - (checksum), ki jo dobimo, če seštejemo vse oktete v posameznem datagramu. Če se vsoti, izračunani na pošiljateljski in prejemniški strani, razlikujeta, potem prejemniški TCP sklepa, da je prišlo med prenosom do napake. Tak datagram se na prejemniški strani zavrže in od TCP na pošiljateljski strani se zahteva ponovno pošiljanje istega

datagrama.



Slika 3.34: Glava TCP datagrama

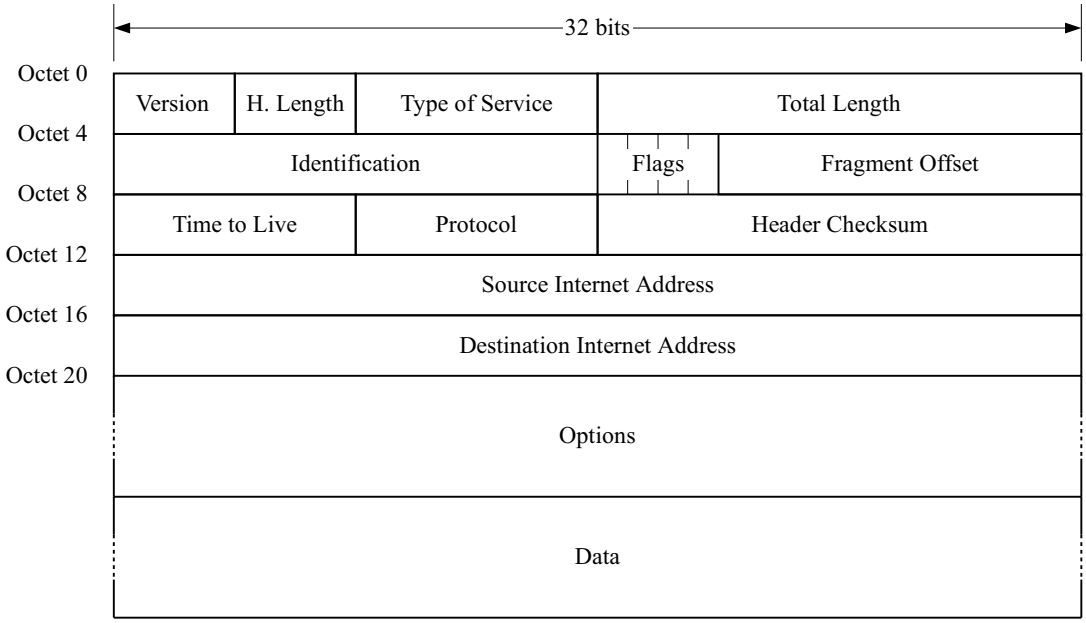
Ostali deli glave datagrama, ki zgoraj niso bili omenjeni v glavnem urejajo spojitev. Zato, da pošiljatelj ve, da je nek datagram prispel do prejemnika, mu mora le-ta poslati neke vrste potrdilo. To potrdilo je datagram, ki ima zapolnjeno polje, ki je na sliki 3.34 označeno kot potrditveno število (acknowledgement number). Če potemtakem prejemnik pošlje datagram s potrditveno številko 1.500, to pomeni da je sprejel vse podatke do okteta z zaporedno številko 1.500. Če pa pošiljatelj po preteku določenega časa od prejemnika ne dobi potrdila o prejetih datagramih, začne ponovno pošiljati podatke. Ker bi bilo po eni strani seveda povsem nepraktično čakati na potrdilo vsakega datagrama pred oddajo naslednjega, pa po drugi strani spet ne moremo neprestano pošiljati datagramov, ker bi lahko računalnik na pošiljateljski strani prehitro pošiljal podatke počasnejšemu računalniku na sprejemni strani, tako da le-ta ne bi utegnil vseh podatkov prejeti. Tako vsak prejemnik pošiljatelju sporoči, koliko oktetov je trenutno sposoben absorbirati in ta podatek spravi v okno (window) glave TCP datagrama. Ko računalnik sprejema podatke se namreč število v oknu manjša in ko se zmanjša na nič, mora pošiljatelj prekiniti pošiljanje. Ko prejemnik podatke obdela, število v oknu ponovno poveča in s tem pošiljatelju sporoči, da je sposoben sprejeti nove oktete.

Pogosto se lahko isti datagram, ki ga prejemnik vrne pošiljatelju, uporabi za potrditev s stališča prejemnika sprejetih datagramov in določitev števila oktetov, ki jih prejemnik lahko še sprejme.

Glava IP (Internet Protocol) datagrama

TCP protokol iz transportnega nivoja pošlje vsak datagram, ki ga je pripravil za prenos preko omrežja, protokolom medmrežnega nivoja, kjer se obdelave datagrama loti IP protokol. Ker je naloga IP protokola izključno, iskanju poti datagrama in prenos na sprejemno stran, od TCP protokola ne potrebuje razen IP naslova prejemnika ničesar drugega. Najpomembnejši podatki v glavi, ki jo prejetemu datagramu z dodano glavo TCP protokola, doda IP protokol, so naslov pošiljatelja (source Internet address), naslov prejemnika (destination Internet address), številka protokola (protocol) in še kontrolna vsota v glavi (header checksum) sporočila. Naslov pošiljatelja in prejemnika sta 32-bitni števili, o katerih smo že nekaj povedali. Številka protokola v glavi IP datagrama sporoča sprejemiku, kateremu protokolu transportnega nivoja (TCP ali UDP) mora dostaviti ta datagram. Običajno je

pri prenosu podatkov TCP tisti protokol, ki preda datagram IP protokolu na oddajni strani, IP pa ga na prejemniški strani vrne TCP protokolu. Kontrolna vsota IP datagrama omogoča IP protokolu na sprejemni strani, da preveri, ali glava ni bila poškodovana ob prenosu, saj lahko v nasprotnem primeru IP pošlje sporočilo napačnemu sprejemniku.



Slika 3.35: Glava IP datagrama

Zastavice (flags) v glavi IP datagrama se uporabljajo za nadziranje odlomkov (fragments) datagramov v primerih, ko moramo sam datagram pri prenosu skozi mrežo zaradi njegove velikosti razdeliti. Življenjski čas (time to live) je število, ki pojema s časom prenašanja datagrama skozi sistem. Ko se to število zmanjša na nič, se datagram odslovi, tako da se v praksi pri premajhnem nastavljenem življenjskem času kaj lahko zgodi, da se posamezni datagrami pri velikem številu prehodov skozi omrežja enostavno izgubijo.

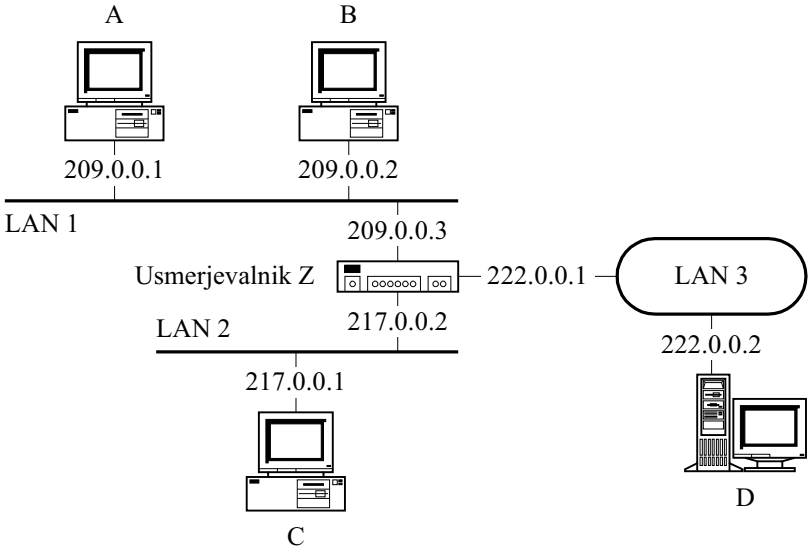
3.8.3 Segmentacija

Iz povsem praktičnih razlogov lahko posamezne mreže predpišejo zgornjo mejo velikosti prenašanega paketa. IP segmentacijska funkcija, ki omenjeni proces izvrši, je torej odgovorna za razstavljanje velikih datagramov na manjše, tako da lahko le-ti glede na fizične omejitve omrežij nemoteno prehajajo od pošiljatelja do prejemnika.

3.8.4 Usmerjanje

Postopek prenašanja datagramov med pošiljateljem in prejemnikom, za katerega je v veliki meri odgovoren IP protokol, imenujemo usmerjanje (routing). Izjemno pomembno za razumevanju delovanja mehanizma usmerjanja v TCP/IP protokolih je dejstvo, da gre pri prenosu podatkov oziroma paketov med pošiljateljevim in prejemnikovim računalnikom, v najbolj splošnem primeru dejansko za dvofazni prenos. Usmerjevalni protokoli, ki jih poganjajo usmerjevalniki, prenesejo namreč najprej paket le do usmerjevalnika na prejemnikovi lokalni mreži, le-ta pa kasneje paket dostavi prejemniku. Vsak uporabniški računalnik ima zato svojo usmerjevalno tabelo, ki jo uporablja za določitev najboljšega sledečega naslova na poti datagrama od pošiljatelja do prejemnika. Taka tabela mora imeti najmanj en vhod, ki vsebuje naslov privzetega usmerjevalnika.

Kako poteka prenos podatkov med uporabniki, si oglejmo na primeru preprostega interneta s slike 3.36, kjer so preko enega usmerjevalnika med seboj povezane tri lokalne mreže.



Slika 3.36: Primer preprostega interneta

Prenos datagrama po lokalni mreži

Zanima nas torej, kako poteka prenos datagrama od uporabnika A do uporabnika B na sliki 3.36. Ko IP protokol pošiljatelja sprejme datagram, razišče prejemniški naslov, ki se nahaja v glavi datagrama. Nato primerja identifikacijsko številko mreže v prejemniškem naslovu IP datagrama z identifikacijsko številko mreže uporabnika B. Če se ti dve vrednosti ujemata, tako kot se v našem primeru, se IP datagram naslovi uporabniku na isti lokalni mreži, kar z drugimi besedami pomeni, da je IP funkcija uporabnika A sposobna prenesti IP datagram uporabniku B neposredno. Pri neposrednem prenosu od uporabnika A do uporabnika B, za katerega je uporabnik A določil, da se nahaja na njegovi lokalni mreži, sproži IP proces uporabnika A vpogled v lastno ARP skladišče. Če v njem obstajajo vrata, ki odgovarjajo IP naslovu uporabnika B, lahko uporabnik A omenjeni datagram takoj prenese prejemniku. Če pa v svojem ARP skladišču uporabnik A ne najde fizičnega naslova uporabnika B, vsem uporabnikom lokalne mreže pošlje zahtevo po identifikaciji. Svoji zahtevi, zgolj zaradi preprečevanja porasta prometa na omrežju, uporabnik A doda svoj fizični naslov. Ko uporabnik A dobi odgovor na svojo zahtevo, v katerem je v našem primeru tudi fizični naslov uporabnika B, lahko začne z neposrednim prenosom datagrama.

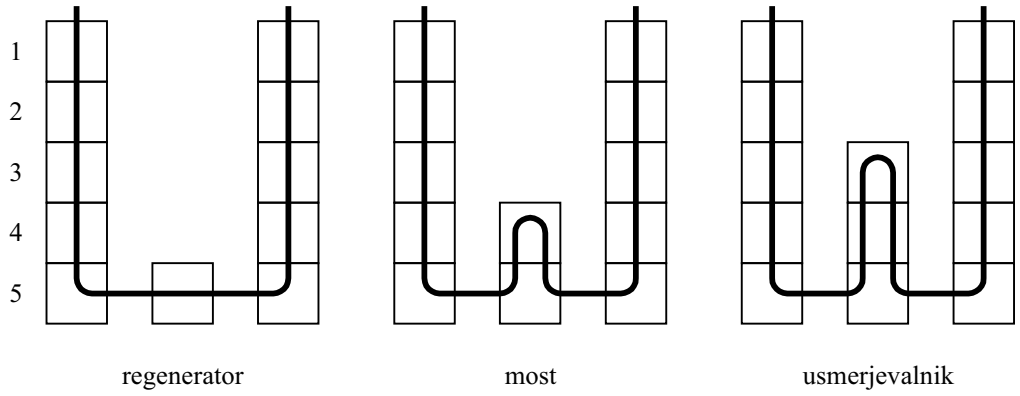
Prenos datagrama med lokalnimi mrežami

V nadaljevanju si oglejmo prenos datagrama od uporabnika A do uporabnika D s slike 3.36. Ker se tokrat identifikacijska številka mreže v prejemniškem naslovu IP datagrama in identifikacijska številka mreže uporabnika A med seboj razlikujeta, mora IP pošiljatelja uporabiti lastno usmerjevalno funkcijo za pošiljanje IP datagrama usmerjevalniku na lokalni mreži, ki ji le-ta pripada. IP proces A-ja poišče vhod na njenegovi usmerjevalni tabeli, ki bi povezoval identifikacijsko številko mreže prejemniškega naslova. V tem primeru usmerjevalna tabela nima ustreznega vhoda, tako da se le-ta poveže s privzetimi vrati oz. usmerjevalnikom. Za specifikacijo naslova strojne opreme usmerjevalnika, gostitelj A uporabi lastno ARP skladišče in pošlje paket do usmerjevalnika. Sedaj ta izlušči identifikacijsko številko mreže, kamor je bil paket namenjen, njegova IP funkcija pa glede na to vrednost izbere ustrezno mrežo.

Usmerjevalnik se še enkrat poveže z lastnim ARP skladiščem, s pomočjo katerega določi naslov strojne opreme naslovnika in nazadnje prenese IP datagram direktno do uporabnika D.

3.8.5 Povezave med lokalnimi mrežami

Če sta tako pošiljatelj kot tudi prejemnik sestavna dela iste lokalne mreže (LAN - Local Area Network), lahko njuna medsebojna komunikacija poteka brez kakršnihkoli vmesnih naprav. Težave pa nastopijo, kadar hoče pošiljatelj poslati sporočilo prejemniku, ki je priključen na drugo lokalno mrežo, saj morajo datagrami pri tem skozi celo vrsto naprav, ki povezujejo več lokalnih mrež in med katerimi so najpogostejši regeneratorji, mostovi in usmerjevalniki. Vse omenjene naprave se med seboj razlikujejo v prvi vrsti po nivojih TCP/IP modela v katerem pride do povezave.



Slika 3.37: Povezovalne mrežne komponent

Regenerator (repeater)

Najenostavnejša naprava za povezovanje lokalnih omrežij je regenerator, ki deluje v fizičnem nivoju ISO-OSI modela oziroma nivoju strojne opreme TCP/IP modela. Regeneratorji, ki povezujejo skupine vodnikov najrazličnejših izvedb in skrbijo za zagotavljanje zahtevane kvalitete signala na njih, brezpogojno preslikujejo podatkovne bite iz svojega vhoda na izhod, pri čemer včasih izgubijo preslikane bite iz začetka ali konca paketov. Teh napak pa sami regeneratorji ne ugotavljajo in popravljajo, saj kot že povedano, delujejo izključno v nivoju strojne opreme TCP/IP modela. Regeneratorji so neodvisni od TCP/IP protokolov, ki jih omrežja poganjajo, in kot taki nezaznavni za končna vozlišča.

Most (bridge)

Most deluje v podatkovnem nivoju ISO-OSI modela oziroma v vmesniškem nivoju TCP/IP modela in pogojno premika pakete iz ene lokalne mreže v drugo. Pakete obdeluje po principu shrani in oddaj, tako da je prenos podatkov preko mostov precej daljši kot pri regeneratorjih. Le-ta na svojem vhodu sprejeme cel paket in ga preda vmesniškemu nivoju, kjer se preveri kontrolna vsota in se opravijo manjše spremembe na paketih. Nato se paket pošlje nazaj na fizični nivo in naprej v mrežo, na kateri se nahaja prejemnik. Ker so mostovi naprave, ki sodijo v vmesniški nivo, ne spreminjajo glav datagramov iz nivojev, ki ležijo višje. Podobno kot regeneratorji so tudi mostovi neodvisni od protokolov, za razliko od njih pa so programabilni.

Usmerjevalnik (router)

Za razliko od mostov, delujejo usmerjevalniki v omrežnem nivoju ISO-OSI modela oziroma v medmrežnem nivoju TCP/IP modela. To jim omogoča, da povezujejo med seboj arhitekturno zelo različne

mreže, cena, ki jo morajo za to plačati, pa je manjša hitrost usmerjanja datagramov. Usmerjevalniki za razliko od obeh zgoraj omenjenih komponent mrež, regeneratorjev in mostov, niso neodvisni od protokolov in posledično niso nezaznavni za računalnike na mreži, saj mora, če hoče nek računalnik poslati datagram skozi usmerjevalnik, potemtakem datagram nasloviti na njegov fizični naslov. Še ena pomembna lastnost usmerjevalnika je, da sam poganja svoje lastne usmerjevalne protokole, ki samostojno iščejo naslednji vmesni ali končni naslov, na katerega morajo datagrami priti.

V splošnem obstajata dve vrsti usmerjevalnikov. Eni zagotavljajo storitev s spojem, drugi pa storitev brez spoja. Ker se v TCP/IP mrežah najpogosteje uporabljajo usmerjevalniki, ki zagotavljajo storitev brez spoja, bomo v nadaljevanju opisali le usmerjevalnik brez spoja.

Kot že vemo, je pri prenosu dolge datoteke potrebno to datoteko razdeliti na posamezne kose. Vsakemu takemu kosu protokoli tansportnega in medmrežnega nivoja dodajo svoje glave. Ker datagram potuje skozi več mrež, morajo protokoli datagramu v vsaki mreži dodati še tretjo glavo, ki je specifična za mrežo v kateri se datagram trenutno nahaja (če je to Ethernet, ima datagram obliko EIT...).

Naloge usmerjevalnikov so:

- premikanje datagramov preko vmesnikov,
- sprejemanje in procesiranje paketov, ki vsebujejo informacije za usmerjanje navadnih datagramov,
- opravljanje raznih nadzornih funkcij.

Usmerjevalniki sprejemajo tiste datagrame, ki so poslani neposredno nanje preko omrežnih vmesnikov. Ko je datagram sprejet, se najprej preveri njegova kontrolna vsota, nato se mu sname vodilni zapis tiste mreže, s katere je ravnokar prispel, programska oprema v usmerjevalniku pa mu nazadnje pregleda ostale glave, da ugotovi kam je datagram namenjen. Če se naslov datagrama ujema z naslovom usmerjevalnika, potem datagram vsebuje informacijo namenjeno samemu usmerjevalniku, v nasprotnem primeru pa je namenjen drugemu prejemniku. Ko usmerjevalnik prejeme datagram, ki ni namenjen njemu, pregleda njegov vodilni zapis ali ta vsebuje točno določeno pot od pošiljatelja preko usmerjevalnikov do prejemnika, saj v TCP/IP protokolih lahko sam pošiljatelj natančno določi pot, po kateri hoče prenesti sporočilo. Takšnemu načinu prenosa datagramov pravimo izvorno usmerjanje (source routing). Če pot datagrama ni vnaprej natanko določena, skuša usmerjevalnik ugotoviti, kam je najprimerneje poslati datagram, saj je datagram lahko namenjen računalniku, ki je priključen na isto mrežo kot sam usmerjevalnik. Če je temu tako, je datagram poslan na naslov prejemniškega računalnika. Usmerjevalnik pa ima za oddajanje datagramov v svojih usmerjevalnih tabelah lahko določeno tudi privzeto pot (default path), na katero odda datagram ne glede na ustreznost njegovega naslova. Zadnja možnost, ki se lahko pri prenosu datagramov med usmerjevalnikom in drugimi mrežnimi komponentami dogodi je, da pot do naslednje mrežne komponente ni znana in da za datagram ni predvidena privzeta pot. V tem primeru usmerjevalnik zavrže sprejeti datagram in pošlje pošiljatelju sporočilo, da je prišlo pri usmerjanju do napake.

Pomembna naloga, ki jo morajo usmerjevalniki opravljati je tudi razdeljevanje TCP/IP datagramov v manjše pakete za prenos po tistih mrežah, ki originalnih TCP/IP datagramov ne morajo prenašati. Ta postopek je s stališča uporabnikov TCP/IP protokolov seveda popolnoma neopazen, kar pomeni, da so vsi manjši paketi poslani na isti naslov usmerjevalnika, ki jih znova sestavi v datagrame.

Usmerjevalna funkcija IP protokola je navadno dovolj močna, da omogoča posameznim gostiteljskim računalnikom vzdrževanje lastnih usmerjevalnih tabel. Ker pa morajo usmerjevalniki sodelovati med seboj, če želijo uspešno prenašati datagrame med posameznimi lokalnimi mrežami, morajo le-ti imeti širši vpogled v svoje usmerjevalne tabele kakor je to potrebno za posamezni gostiteljski računalnik. Zato pogosto usmerjevalniki poleg IP protokola uporabljajo svoje usmerjevalne protokole, ki jim omogočajo, da so njihove usmerjevalne tabele čimbolj točne.

3.9 Porazdeljeni sistemi

Porazdeljeni sistemi (distributed systems) so kljub temu, da je bila ideja porazdeljenih sistemov pred leti skoraj zavrnjena, dandanes že množično v uporabi, saj je niz tržnih in uporabniških produktov zasnovanih prav na njihovi osnovi.

Porazdeljena obdelava podatkov (DDP - Distributed Data Processing) je določena kot obdelava podatkov, pri kateri:

- je obdelava namesto na enem mestu organizirana okoli več procesnih elementov (PE), ki so lahko računalniki ali drugi sistemi sposobni izvajanja avtomatskih in inteligentnih operacij,
- so PE lahko organizirani na funkcionalni ali geografski osnovi in sodelujejo pri podpori zahtevam uporabnika,
- je povezovanje PE izvedeno preko skupnih nosilcev ali privatnih povezav,
- je dopuščena porazdelitev materialne in/ali programske opreme in/ali podatkov posameznih PE.

3.9.1 Porazdelitev in decentralizacija

Porazdeljeno procesiranje omogoča funkcionalno in geografsko porazdelitev, kjer so porazdeljeni deli integrirani v celovit in skladen (koherenten) sistem.

Uporaba DDP je posledica hitre rasti osebnih računalnikov in povezovanja le-teh v sisteme. Trend uporabe DDP je najmočnejši v mikroelektroniki in v podatkovnih ter multimedijskih komunikacijah, kateri tipični primeri uporabe so telefonske rezervacije vozovnic, preverjanje bančnega stanja ipd.

DDP zahteva enovito organiziranost v sistemu, ki jo uporablja.

3.9.2 Prednosti in slabosti porazdeljenih sistemov

Prednosti

- Bistvena prednost uporabe DDP je zmanjševanje stroškov, saj lokalna obdelava podatkov navadno zmanjša promet po komunikacijskem omrežju, podatke pa lahko vnašamo in popravljamo na različnih mestih. Porazdeljeni sistemi omogočajo zmanjševanje stroškov tudi pri uporabi cenejše in manj zahtevne strojne opreme zaradi manjše kompleksnosti določenih aplikacij, pogosto pa omogočajo tudi boljšo izkoriščenost sistema s strani uporabnikov.
- Izboljšanje odzivnega časa sistema je prav tako pomembno pri uporabi DDP, saj smo običajno pri centraliziranih in preobremenjenih sistemih in počasnih linijah doživljali velike zamude pri obdelavi podatkov. Lokalna obdelava zmanjšuje odzivnost sistema oziroma čas celovite obdelave, tako da se lahko posamezne funkcije izvršujejo vzporedno na več funkcionalno porazdeljenih računalnikih. Vzporedna obdelava podatkov je postala dominantna v DDP sistemih.
- Če želimo shraniti varnostne kopije podatkov v velikem centraliziranem sistemu, običajno potrebujemo še eno veliko, prav tako centralizirano računalniško enoto. Ta pristop je dokaj neprimeren, saj moramo vzdrževati kompatibilnost med s shranjenimi podatki na vseh nivojih obeh sistemov. Po drugi strani pa porazdeljeni sistemi omogočajo učinkovito in ceneno shranjevanje podatkov, saj izpad vsakega dela, ki je integriran v celoto, pomeni le prenos njegovih opravil na druge računalnike.

Slabosti

- Dokaj realno lahko rečemo, da porazdeljeni sistemi niso namenjeni vsakomur, saj obstaja v DDP sistemih velika nevarnost izgube nadzora nad sistemom.
- Naslednja slabost DDP je v podvajanju programskih paketov v sistemu, kar se odraža v nesmiselno visoki ceni programske opreme. Podvajanje podatkov pa pogosto povzroča tudi konflikte med posameznimi komponentami DDP sistemov .
- Če porazdeljeni uporabniki želijo dodajati podatke v skupno podatkovno bazo, morajo najprej narediti kopijo teh podatkov, ki jo kasneje pridajo izvirnim podatkom kot kopijo.
- Prav tako lahko pogosto v DDP sistemih nastopajo zaradi nekompaktibilnosti med posameznimi modeli problemi z materialno opremo. Vzdrževanje na daljavo lahko povzroča velike težave posebno pri hitri odpravi tehničnih napak.
- Nezdružljivost strojne in programske opreme, do katere pride največkrat pri komunikacijskih protokolih in programskih paketih za osebne računalnike. Kot posledica teh težav je bila razvita cela kopica emulacijskih programskih paketov.

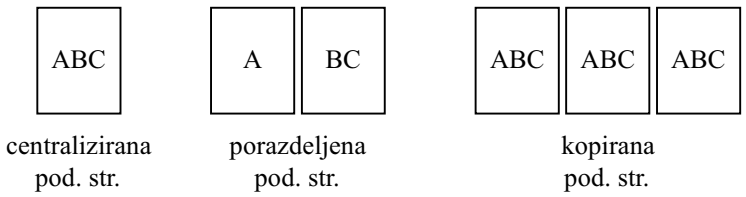
3.9.3 Porazdeljene podatkovne baze

Porazdeljene podatke baze predstavljajo zanimiv problem pri porazdeljeni obdelavi podatkov, pri katerem pa je potrebno biti posebej pazljivi, da bi lahko izrabili potencialne prednosti.

Tipi porazdeljenih podatkovnih baz

Poznamo tri osnovne tipe porazdelitev podatkov in sicer:

- centralizirane podatkovne baze, ki dejansko ne predstavljajo porazdeljene podatkovne baze. V tem primeru se podatki nahajajo na enem mestu, kjer se vrše tudi vsi posegi iz oddaljenih terminalov.
- porazdeljene podatkovne baze, kjer podatke razbijemo na kose in jih vgradimo v določene podgrupe v omrežju. Vse porazdeljene podatkovne baze lahko imajo isto strukturo, obliko ali metodo dostopa do podatkov. V tem primeru govorimo o homogeni podatkovni bazi. Če pa porazdeljeno bazo sestavljajo različne oblike, strukture ali metode dostopa, tedaj govorimo o heterogeni podatkovni bazi.
- kopirane podatkovne baze na raznih lokacijah pa predstavljajo tretji tip baze podatkov. Ta tip predstavljajo večkratne enake kopije baze na posameznih mestih oziroma delih mreže. Enake podatkovne strukture med kopijami podatkov imenujemo homogene večkratne strukture podatkov. V primeru, da pa kopije baze podatkov preoblikujemo ali uporabimo različne metode dostopa, tedaj govorimo o heterogenih večkratnih strukturah podatkov.



Slika 3.38: Tipi porazdeljenih podatkovnih baz

Nadalje lahko razdelimo baze podatkov po vertikalni ali horizontalni porazdeljenosti. Pri vertikalni porazdelitvi so podatki hierarhično porazdeljeni. Podrobnejši podatki se lahko shranijo lokalno, pomembnejši podatki pa se shranijo višje v strukturi. Horizontalna porazdelitev dejansko porazdeljuje podatke po delovnih skupinah (oziroma njihovih potrebah) v omrežju.

3.9.4 Koncepti in izrazi v porazdeljenih podatkovnih bazah

Enovito (konsistentno) stanje. Vsi podatki v mreži so točni in pravilni. Enaki podatki se nahajajo v osnovni podatkovni bazi kot tudi v vseh njenih kopijah.

Prenos (transakcija) je sekvenca operacij, ki pretvarja trenutno enovito stanje v novo enovito stanje. Tak je na primer bančni prenos denarja z enega na drug račun, ko baza preide iz enega v drugo enovito stanje.

Ker nekateri podatkovni sistemi omogočajo neomejeno število prenosov oziroma operacij, tovrstne sisteme težko nadziramo in zato se tovrstnim večfunkcijskim operacijam izogibamo.

Začasna neenovitost (nekonsistentnost). Ta izraz uporabljamo pri opisu faze izvajanja neke operacije. Vzemimo, da prenašamo 100 SIT med dvema uporabnikoma; ta operacija predstavlja začasno neenovitost podatkovne baze.

Vzemimo primer programa

BEGIN

TI - 1 - branje stanja računa A
TI - 2 - branje stanja računa B
TI - 3 - vpiši na račun A $\leftarrow$ stanje A $-$ 100 SIT
TI - 4 - vpiši na račun B $\leftarrow$ stanje B + 100 SIT

END

Po prvi operaciji pisanja v bazo podatkov so podatki začasno neenoviti. Le po popolnem dokončanju programa lahko govorimo zopet o enoviti bazi.

Spor (konflikt). Po zaključku niza operacij se dogodi, da je rezultirajoče stanje baze neenovito. Do konflikta pride, če istočasno izvajamo dve ali več operacij nad istim podatkom. Oglejmo si konflikt na primeru z dvema procesoma T1 in T2:

BEGIN

T1 - 1 - branje stanja računa A
T2 - l - branje stanja računa B
T1 - 2 - branje stanja računa B
T1 - 3 - vpis na račun A $\leftarrow$ stanje računa A $-$ 100 SIT
T2 - 2 - beri stanje računa B
T1 - 4 - vpiši na račun B $\leftarrow$ stanje računa B + 100 SIT
T2 - 3 - vpiši na račun A $\leftarrow$ stanje računa A $-$ 75 SIT
T2 - 4 - vpiši na račun B $\leftarrow$ stanje računa B + 75 SIT

END

Rezultat operacije je neenovita podatkovna baza, saj je med izvajanjem obeh prenosov prišlo do spora. Podrobnejša analiza programa in obeh transakcij T1 in T2 prikazuje v tabeli 3.3 dejanske podledice konflikta.

Transak-cija	Delovni prostor transakcije T1		Delovni prostor transakcije T2		Podatkovne baze	
	Rač A	Rač B	Rač A	Rač B	Rač A	Rač B
T1-1	300	—	—	—	300	300
T2-1	300	—	300	—	300	300
T1-2	300	300	300	—	300	300
T1-3	200	300	300	—	200	300
T2-2	200	300	300	300	200	300
T1-4	200	400	300	300	200	400
T2-3	200	400	225	300	225	400
T2-4	200	400	225	375	225	375

Tabela 3.3: Primer neenovite podatkovne baze

Razvrščanje (scheduling) predstavlja določanje vrstnega reda ali kode dogodkov. Predhodni primer kaže na slabo razvrščanje, saj se sporu lahko izognemo le, če se vsaka transakcija vrši samostojno, šele nato pa se prenese v skupno bazo.

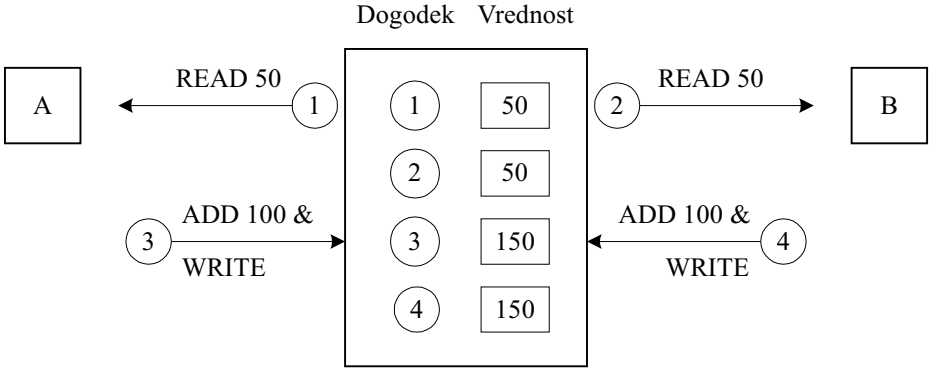
Zaporedno razvrščanje (sequential scheduling) je razvrščanje z enakim učinkom, kot je razvrščanje operacij večih prenosov. Rezultat zaporednega razvrščanja je vedno konsistentno stanje.

Zaklepanje (locking) podatkov zagotavlja nedostopnost do podatkov v fazi neenovitosti podatkovne baze. Uporablja se kot zaščita v sistemu z večimi prenosi, saj se z uporabo zaklepanja lahko izognemo konfliktnemu stanju. Upravljalni sistem podatkovne baze (DBMS - Data Base Management System) je odgovoren za zaklepanje aktivnih podatkov.

Prožnost (resiliency) je zmožnost vzpostavitve prvotnega stanja v porazdeljenih podatkovnih bazah, ko pride do izpada sistema. Tedaj se zaradi nepopolno izvršenih transakcij povrnemo nazaj v začetno stanje in restavriramo bazo v začetno obliko.

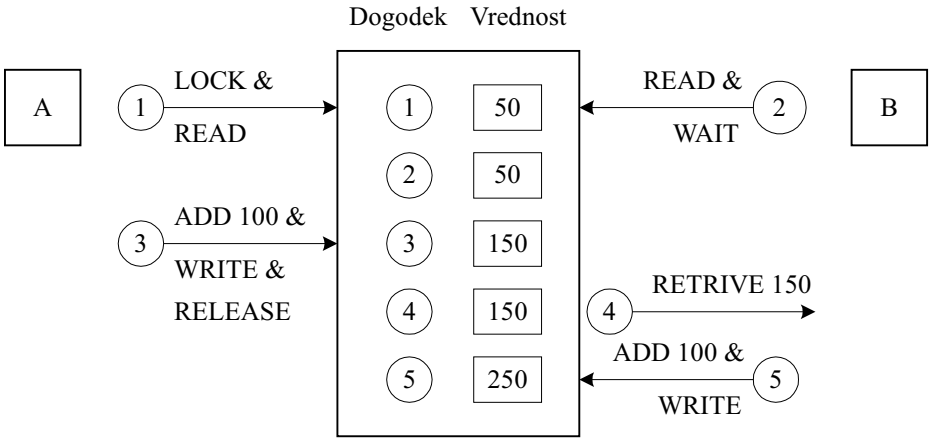
3.9.5 Primer uporabe porazdeljenih podatkovnih baz

Porazdeljeni podatki ali sistemi so vedno zahtevni sistemi. Vzemimo primer s slike 3.39, kjer uporabika A in B simultano dopolnjujeta posamezne podatke v podatkovni bazi. V primeru odsotnosti nadzornega mehanizma podatkovna baza izvede le en postopek dopolnjevanja, drugi postopek pa je izgubljen. To se dogodi, ko oba uporabnika posegata po podatku, ga spremenita in vpišeta spremenjeno vrednost v bazo.



Slika 3.39: Hkratni dostop uporabnikov do porazdeljene podatkovne baze

Omenjeni problem rešimo z zaporednim razvrščanjem dogodkov, ki je prikazano na sliki 3.40.



Slika 3.40: Zaporedni dostop uporabnikov do porazdeljene podatkovne baze

Poglavje 4

Vzporedni računalniški sistemi

4.1 Klasifikacija - razvrstitev vzporednih računalniških sistemov

V bistvu razvrščamo vzporedne računalnike v tri kategorije in sicer po treh različnih kriterijih:

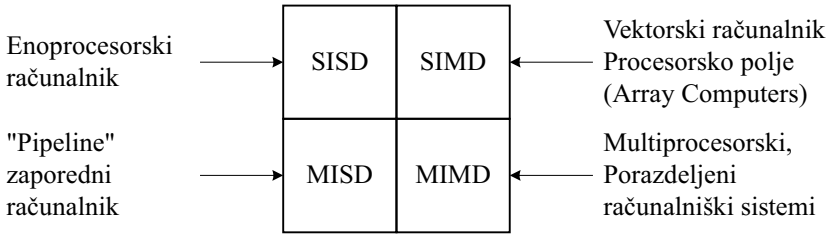
- glede na strukturo,
- glede na njihov nivo abstrakcije in
- glede na vzporedne operacije klasificirane po tipu argumenta.

4.1.1 Sistemska razvrstitev

Po Flynnu lahko računalnike razvrstimo v štiri kategorije:

- SISD (Single Instruction, Single Data),
- SIMD (Single Instruction, Multiple Data),
- MISD (Multiple Instruction, Single Data),
- MIMD (Multiple Instruction, Multiple Data).

Najzanimivejša za uporabo v telekomunikacijah sta SIMD (sinhroni paralelizem) in MIMD (asinhron paralelizem). SISD struktura predstavlja razred von Neumanovih arhitektur, MISD razred pa pokriva "zaporedne" "pipeline" strukture.



Slika 4.1: Razvrstitev računalniške arhitekture po Flynnu

Sinhroni paralelizem pomeni, da obstaja le en sistem nadzora. V tem primeru posebni procesor izvršuje program, med seboj povezani procesorji z enostavno strukturo pa pod nadzorom tega procesorja izvršujejo posamezne ukaze po korakih (sinhrono).

Asinhroni paralelizem pa zahteva več mest nadzora. V tem primeru vsak procesor izvaja svoj del programa ali kar svoj program. Za izmenjavo podatkov je v tem primeru potrebno poskrbeti ustrezno

sinhronizacijo med procesorji. Po tej razdelitvi lahko oba razreda MIMD in SIMD razdelimo še na dva podrazreda in sicer glede na metode povezovanja procesorjev.

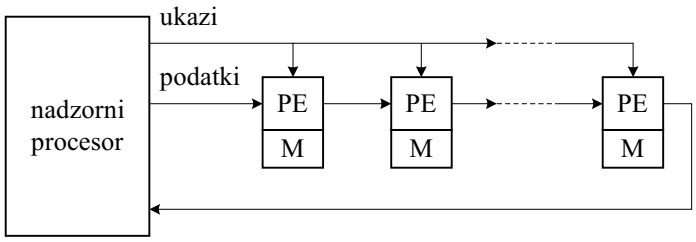
MIMD

- Povezava preko porazdeljenega pomnilnika (tesno povezani)
  → Multiprocesorski sistem
- Povezave preko povezovalnega omrežja z uporabo sporočil
  → Porazdeljeni računalniški sistemi

SIMD

- Brez globalnih povezav - samo verižne povezave med procesorji
  → Vektorski računalniki
- Povezava preko mreže
  → Procesorska polja

Enostavna zaporedna - "pipeline" struktura je prikazana na sliki 4.2.



Slika 4.2: Enostavna zaporedna (verižna) povezava med procesorji

Porazdelitev procesorjev omogoča zaporedno izvajanje ukazov. Sekvenčno ali vzporedno/zaporedno izvajanje ukazov je predstavljeno na sliki 4.3.

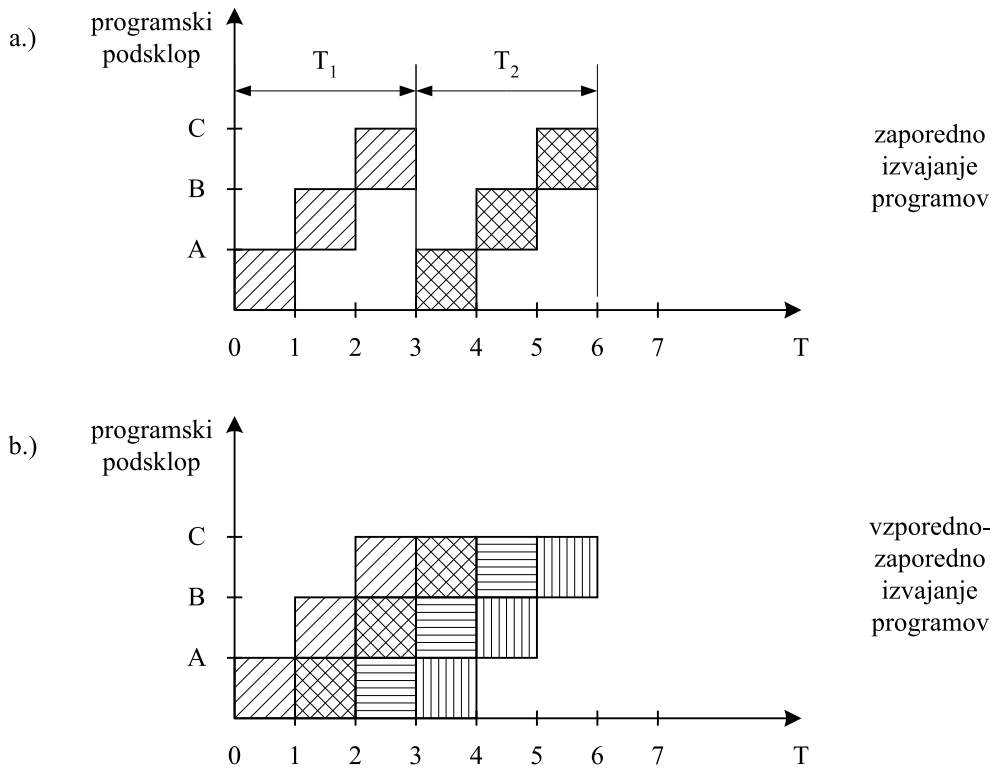
Osnovni program je razdeljen na tri podsklope in sicer: A, B, C. Tak niz ukazov se lahko ponavlja. V primeru sekvenčnega izvajanja se vedno izvajajo le posamezni podsklopi programa, dočim se pri $N - 1 = 2$ (N - stopnja vzporednosti) vzporedno izvajajo vsi podsklopi programa. To pa pomeni, da dobimo rezultat v vsakem ciklu in ne le v vsakem tretjem ciklu. Zaporedno/vzporedne (pipeline) strukture se uporabljajo le za specifične namene.

Enostavna zaporedna računalniška struktura je lahko enoprocesorski sistem z dodano aritmetično logično zaporedno enoto, ki jo prikazuje slika 4.2. V primeru, da je dodanih več neodvisnih zaporednih struktur, govorimo o MIMD sistemu ali o vzporedno/zaporedni strukturi (multiple - pipeline system).

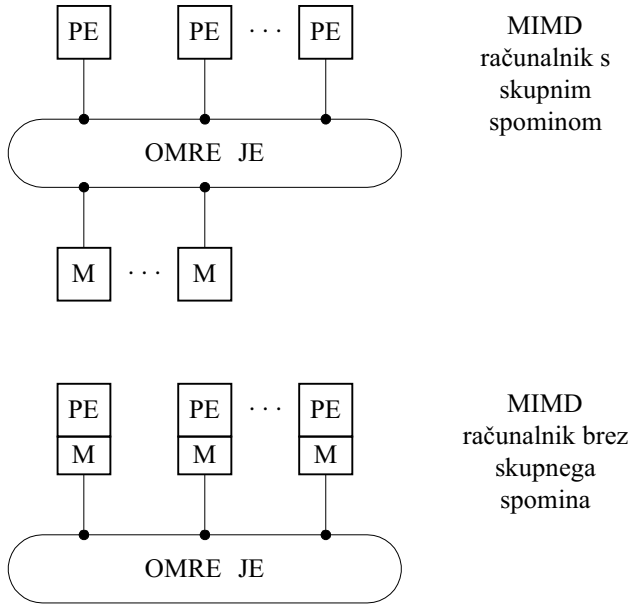
4.1.2 Struktura MIMD

Računalniki tega razreda imajo v primerjavi z računalniki SIMD razreda splošnejšo strukturo in vedno asinhron način delovanja. V tem razredu vsak procesor izvaja svoj program, vsak ima lasten nadzor nad povezovanjem in pretokom podatkov. Pri tem lahko le še razlikujemo razred, kjer imajo računalniki skupni spomin od razreda, kjer imajo računalniki le lastni spomin.

MIMD računalniki s skupnim spominom so znani kot "tesno povezani" sistemi. Sinhronizacija in izmenjava vseh informacij potekata preko skupnega pomnilnika, ki je dosegljiv vsem procesorjem. MIMD računalniki brez skupnega pomnilnika pa so znani kot "šibko povezani" sistemi. Vsaka enota ima lastni spomin, zato predstavljajo niz med seboj povezanih a samostojnih računalnikov. V tem



Slika 4.3: Prikaz računanja v a) enoprocesni sistem, b) v enostavni zaporedni povezavi procesorjev

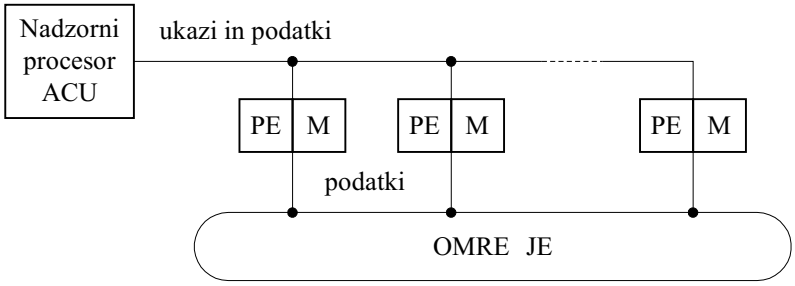


Slika 4.4: MIMD računalnik s skupnim spominom in MIMD računalnik brez skupnega spomina

primeru je postopek sinhronizacije zahtevnejši in dražji, saj mora priti do izmenjave sporočil z uporabo komunikacijskega omrežja.

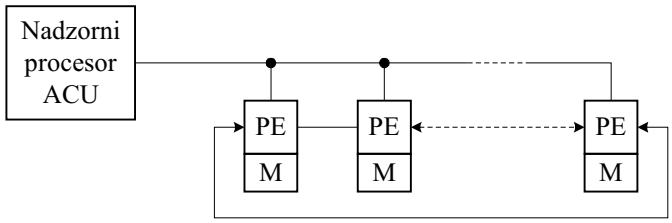
4.1.3 Struktura SIMD

Procesorska polja (Array Computers), za katere se izraz SIMD uporablja kot sinonim, imajo enostavnješo strukturo kot MIMD sistemi. Materialna oprema za razpoznavo in nadzor izvajanja ukazov je realizirana le enkrat in sicer v centralnem nadzornem procesorju, (ACU - Array Control Unit). V tem slučaju je vsak procesor realiziran kot ALE (aritmetično-logična enota) z lokalnim spominom in povezovalno enoto, ki komunicira z ustreznim omrežjem. Ker je v sistemu le en sam dekoder ukazov, poteka izvajanje vedno sinhrono, kar pa pomeni, da obstaja le en nadzorni cikel za cel vzporedni program. Vsaka PE torej izvede isti ukaz znotraj lokalnega spomina, ali pa je neaktivna.



Slika 4.5: Procesorsko polje (Array Computer)

Struktura vektorskega računalnika je enostavnejša od omenjene strukture, saj ne vsebuje globalnega omrežja povezav. Lokalni podatki so zgrajeni kot vektorski register, kjer se operacije vrše nad komponentami vektorja. Enostavna zamenjava podatkov ali pomiki ter rotacije, se običajno izvrše preko posebne podatkovne povezave med PE.



Slika 4.6: Vektorski računalnikški sistem

4.2 Lastnosti mrež in povezovanje procesorjev

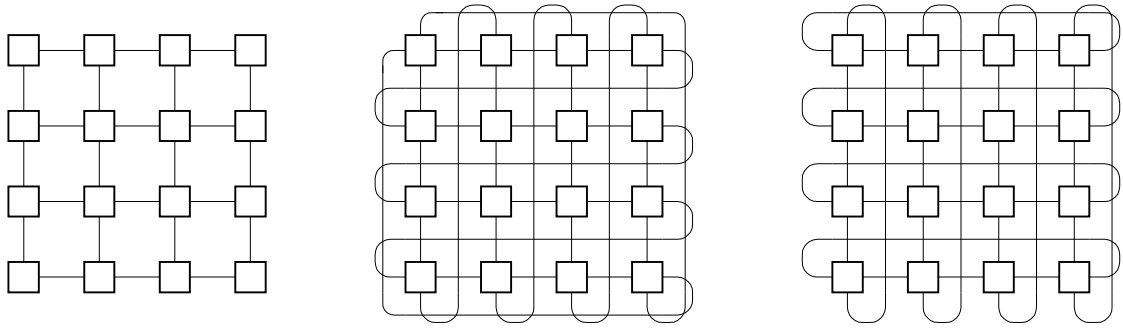
V tem podpoglavju se bomo seznanili s šestimi tipičnimi metodami povezovanja procesorjev v vzporedne računalniške sisteme.

4.2.1 Mreža

Mreža je organizirana v q - dimenzionalne rezine. Osnova mreže je vozlišče. Povezave med njimi so takšne, da so možne le med sosednjimi vozli (slika 4.7). Nekatere mreže omogočajo povezavo med njenimi končnimi vozli "wrap - around". Te povezujejo vozle v isti vrstici ali v istem stolpcu, lahko pa obstoje tudi "toroidne" povezave.

Učinkoviti algoritmi za manipulacijo z matrikami ali algoritmi za razvrščanje uporabljajo strukturo mreže. Ta struktura je tudi uporabna pri reševanju parcialnih diferencialnih enačb drugega in višjih redov.

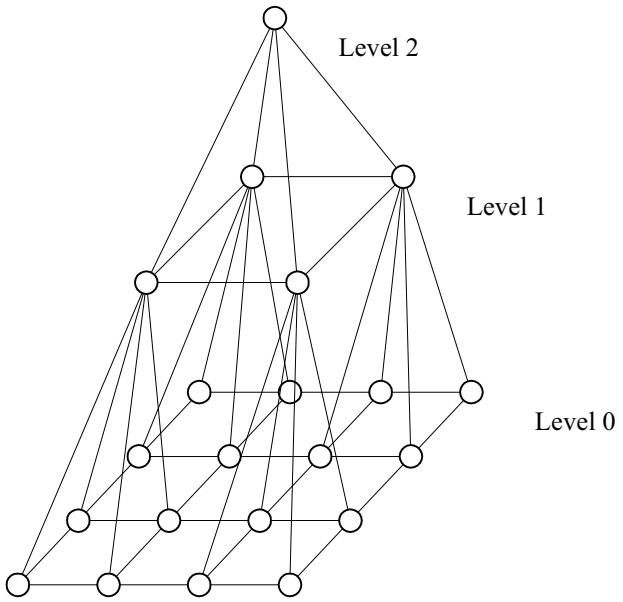
Mrežne povezave imajo tudi določene slabosti kot so počasen prenos podatkov, odsotnost globalnih povezav in nezmožnost spreminjanja njihove strukture.



Slika 4.7: 2D mreže povezav procesorjev

4.2.2 Piramida

Piramidalno povezovalno omrežje velikosti p je 4-krat koreninjeno drevo višine $\log_4 p$, podprtih z dodatnimi medprocesorskimi povezavami. Tak sistem vsebuje na vsakem nivoju drevesa dvodimenzionalno povezovalno mrežo. Tovrstna piramida velikosti p ima na vsakem nivoju $p = k^2$ procesorjev. Celotno število procesorjev piramide velikoti p je $\frac{3}{4}p - \frac{1}{3}$. Nivoji - plasti piramide se označujejo z naraščajočim zaporedjem, pri čemer je najnižji nivo označen z 0, procesor na vrhu piramide pa je na plasti št. $\log_4 p$. Vsak notranji procesor je povezan z devetimi drugimi procesorji: enim očetom, štirimi sosedi iz mreže in štirimi potomci.

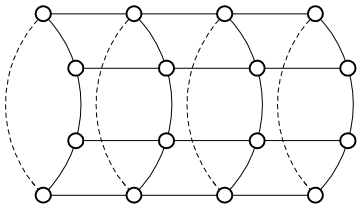


Slika 4.8: Piramidalna struktura povezav med procesorji

4.2.3 Rezine mrež

Graf multiprocesorja z rezino, ki je navadna kvadratna mreža, vsebuje n^2 vozlov, kjer je vsak povezan s svojim sosedom. Za primer $n = 4$ s slike 4.9, kjer je razvidna horizontalna povezava procesorjev (wrap - around) v cilinder.

V tem primeru je procesor $\mathrm{P}(i, j)$ $(i, j \in \{1, \ldots, n\})$ povezan s $\mathrm{P}(i, j-1)$, $\mathrm{P}(i-l, j)$ in $\mathrm{P}(i+l, j)$. Običajno so procesorji z robov povezani ciklično. Tovrstna povezava omogoča določeno sistemsko kot programsko fleksibilnost in reducira premer grafa z $2n - 2$ na $2\left\lfloor \frac{1}{2}n \right\rfloor$.



Slika 4.9: Ciklična mreža (cilinder)

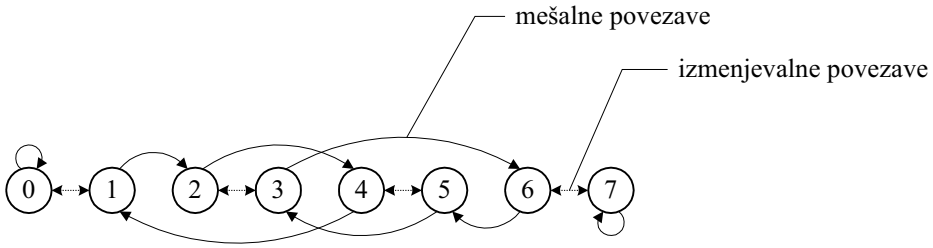
Tovrstna lokalna povezanost za sistem SIMD je lahko razširljiva na sistem, kjer procesorji komunicirajo tudi preko globalnih povezav. Npr. hitra povezovalna vodila lahko omogočajo hitre povezave med npr. vrsticami (procesorji) oziroma med stolpci.

4.2.4 Mešalno - izmenjevalno omrežje

Tovrstno omrežje vsebuje $n = 2^k$ vozlov, označenih z $0, 1, \ldots, n-1$ in dvema vrstama povezav imenovanih mešalne in izmenjevalne povezave.

Izmenjevalne povezave povezujejo pare vozlov, katerih naslovno število se razlikuje le v njihovem "najmanj pomembnem" bitu (least significant bit).

Popolne mešalne povezave povezujejo vozle "i" z vozli "$2 \cdot i$" po modulu $(n-1)$ z izjemo vozla $(n-1)$, ki je tudi povezan sam s seboj.



Slika 4.10: Mešalno izmenjevalno omrežje

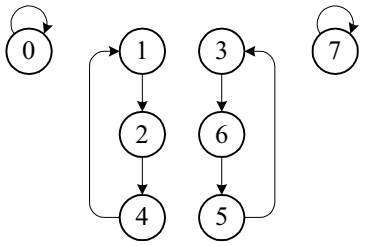
Na sliki 4.10 je prikazano omrežje osmih vozlov z mešalno - izmenjevalnimi povezavami. Zato, da bi bolje razumeli izraz "popolno mešanje", vzemimo osem igralnih kart, oštevilčenih z $0, 1, \ldots, 7$. Če paket razdelimo na dve točni polovici in izvedemo "popolno mešanje", tedaj so karte razvrščene po sledečem redu:

0, 4, l, 5, 2, 6, 3, 7.

Končna lega karte s položaja "i" se lahko določi s popolnim mešanjem v začetkom pri vozlu "i". Vzemimo, da so $a_k, a_{k-1}, a_{k-2}, \ldots, a_1$ binarni naslovi vozla v strukturi mreže s popolnim mešanjem povezav, tedaj je pripadajoča naslednja adresa po operaciji mešanja $a_{k-1}, a_{k-2}, a_{k-3}, \ldots, a_1, a_k$. To pa pomeni, da operacija mešanja nad delom podatkov pomeni le pomik v cikličnem zamiku (rotaciji) naslova za en bit. Če je $n = 2^k$, tedaj k mešalnih operacij pomakne naslov nazaj na sam izvorni naslov. Vozlišča, skozi katera potuje podatek, ki starta na naslovu "i", se imenuje "veriga - i". Ta veriga ni daljša kot log$n$. Veriga, krajša od $\log_2 n$ se imenuje kratka veriga. Primeri teh verig so razvidni s slike 4.11.

4.2.5 Metuljčno povezovalno omrežje (Butterfly network)

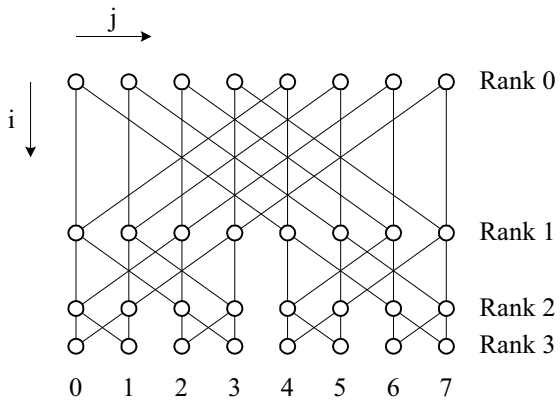
Metuljčno povezovalno omrežje sestavlja $(k+1) \cdot 2^k$ vozlov porazdeljenih v $(k+1)$ vrstico (rank), kjer vsebuje vsaka po $n = 2^k$ vozlov - procesorjev. Vsaka vrstica s slike 4.13 (rank) je označena z 0 do k.



Slika 4.11: Veriga povezav

a_0	a_1	a_2		0	0	1	1
a_1	a_2	a_0		0	1	0	2
a_2	a_0	a_1		1	0	0	4
a_0	a_1	a_2					1

Slika 4.12: Primeri mešanja



Slika 4.13: Metuljčne povezave procesorjev

Vezemimo vozel (i, j), ki pripadaja j-tem vozlu na i-ti vrstici (rank), kjer je $0 \leq i \leq k$ in $0 \leq j \leq n$. Potem je vozel (i, j) v vrstici $i > 0$ povezan z dvema vozloma na nivoju - vrstici $i - 1$: vozel $(i - 1, j)$ in vozel $(i - 1, m)$. Celo število m določimo iz inverzije i-tega najpomembnejšega bita v binarni predstavitvi števila j.

Primer: vozel (3,2)

$j = 2$
$i = 3$

povezave: $(i - 1, j) = (2, 2)$
$(i - 1, m) = (2, 3)$

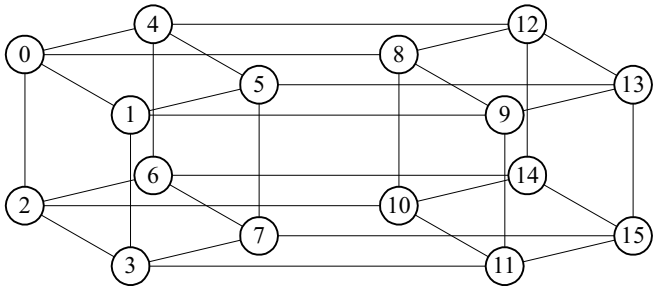
$j = 010 \rightarrow 2$
$m = 011 \rightarrow 3$

Torej, če je vozel (i, j) povezan z vozlom $(i-1, m)$, potem je vozel (i, m) povezan z vozlom $(i-1, j)$, kar predstavlja algoritem za določitev "metuljčnih" povezav.

Celotno vezje je torej sestavljeno iz tovrstnih "metuljčnih" vzorcev, zato tudi tako ime. Čim red - "rank" upada, tedaj se širina "kril" eksponencialno širi.

4.2.6 Hiperkocka

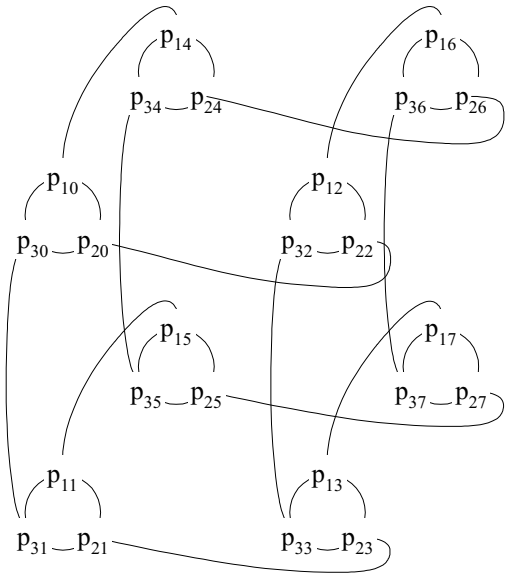
Kockasto povezovalno omrežje je metuljčna povezava, kjer stolpci sovpadajo v enojne vozle. Formalno lahko omrežje sestavlja $n = 2^k$ vozlišč, ki tvorijo k - dimenzionalno hiperkocko. Vozlišča označimo z $0, 1, \ldots, 2^k - 1$. Dva vozla sta sosednja vozla, če se njuna naslova razlikujeta ravno za en sam bit. Štiridimenzionalna hiperkocka s slike 4.14 vsebuje 16 vozlišč.



Slika 4.14: Štiridimenzionalna hiprerkocka

4.2.7 Kocka s cikličnimi povezavami

Kocka s ciklično-povezanimi vozli namesto vozlov v osnovni kocki je predstavljena na sliki 4.15. To je k-dimenzionalna hiperkocka, katere 2^k členov, ki so dejansko krogi k-vozlov, je povezanih vertikalno v metuljčno povezovalno mrežo, katere niza 0 in k sta identična. Pri vsaki dimenziji ima vsak cikel (krog) vozel povezan z vozlom v sosednjem krogu iste dimenzije. Na sliki 4.15 je tako predstavljena 24 vozliščna kocka s krožnimi vozliščnimi povezavami.



Slika 4.15: Kocka s cikličnimi povezavami

Dejansko je vozel (i, j) povezan na vozel (i, m) samo in samo če je m rezultat inverzije i-tega najpomembnejšega bita v binarni predstavitvi števila j. Ta povezava se pomalem razlikuje od predhodno podane metuljčne povezave in sicer: če je vozel (i, j) povezan z vozlom $(i-1, m)$ v metuljčni povezavi, kjer je $j \neq m$, tedaj je vozel (i, j) povezan z vozlom (i, m) v kockasti strukturi s cikličnimi povezavami s slike.

Vsekakor vozel (i, j) še vedno komunicira z vozlom (i, m) z dvema nadaljnima povezavama, tam tudi obstaja direktna vez med vozloma (i, m) in $(i - 1, m)$.

Čisto naravno je, da se postavlja vprašanje, katera izmed omenjenih procesorskih organizacij (mreža, piramida, metulj, popolno mešanje ali kubično usmerjeni cikli) je superiorna za posamične tipe problemov, ali celo, katera od teh organizacij je na splošno najboljša. Nekateri avtorji dokazujejo, da imajo v kocko povezani procesorji optimalno strukturo učinkovitih splošno uporabnih (povezovalnih) sistemov.

4.3 Nekateri primeri multiprocesorskih arhitektur

Arhitekture zasnovane na SIMD modelih, lahko variirajo zaradi:

- števila procesorjev, ki je lahko fiksno ali neomejeno in
- organizacije povezav med procesorji, ki so lahko s skupnim ali porazdeljenim pomnilnikom povezane v mrežo (MC), piramido (P), popolno mešanje (PS), kockasto omrežje (CC - cube connected) ali kockasto- krožno omrežje (CCC - Cube-Cycle Connected) .

4.3.1 Porazdeljen spomin - SIMD model

V literaturi zasledimo niz rešitev SIMD arhitektur s porazdeljenim spominom. Goldschlagerjev model iz leta 1982, imenovan SIMDAG, omogoča večim procesorjem hkratno branje (simultaneous read access) iste pomnilniške lokacije. Prav tako dopušča več kot enemu procesorju, da hkrati zapisuje v isto lokacijo. V primeru hkratnega zapisa, ima procesor z najnižjim indeksom največjo prioriteto.

P-RAM pristop Wyllie-a iz leta 1979 dopušča hkratno branje, ne pa hkratnega pisanja na isto lokacijo. EREW-P-RAM (exclusive read, exclusive write) arhitektura, ki jo je zasnoval Snir leta 1982, dopušča "hkraten" dostop do pomnilniške lokacije le enemu procesorju. CREW-P-RAM (concurrent read) arhitektura dopušča hkraten pristop v ciklu branja večim procesorjem, a prepoveduje pa hkratno pisanje. CRCW arhitektura pa dopušča hkratno branje kot pisanje.

Kljub veliki popularnosti arhitekture s porazdeljenim spominom, ni nihče realiziral realnega superračunalnika na tej osnovi. Mnogo realneje je, da ima vsak procesor lasten spominski modul, pri čemer se podatki zmenjujejo preko povezovalnega omrežja.

4.3.2 Mrežno povezani SIMD model (SIMD - MC)

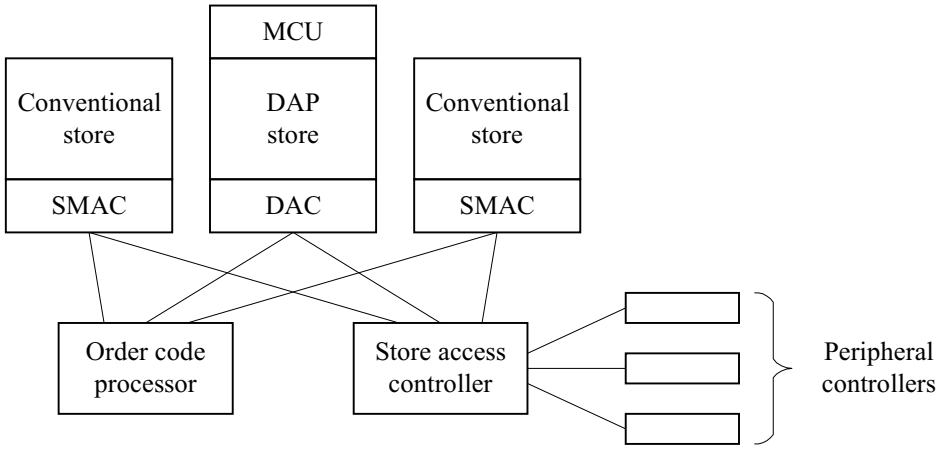
Pri SIMD-MC (Mesh Connected) modelu so procesorji organizirani v q-dimenzionalno rezino. Računalniki kot ILLIAC IV, BOROUGHS PEPE, Esodgear Aerospace MPP in ICL DAP, so spadali v isto kategorijo SIMD-MC arhitekture.

DAP je komercialni vzporedni računalnik zasovan na SIMD-MC modelu. Bitno serijski procesorji reda 64×64 tvorijo mrežo, kjer je vsak procesor povezan s svojim najbljižjim sosedom.

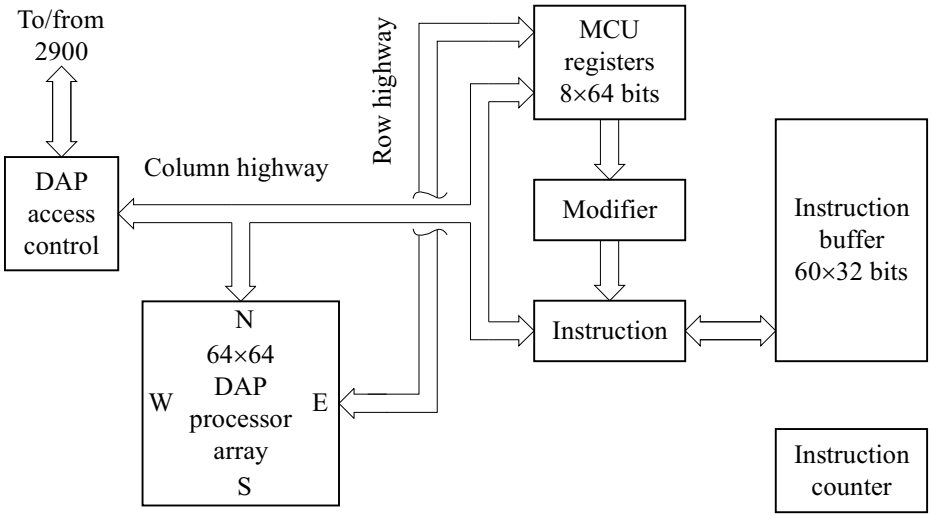
DAP računalnik emulira 2 Mb spominske module, ki imajo sposobnost procesiranja podatkov na povsem vzporeden način. DAP je bil zasnovan kot osnovni modul ICL 2900 računalnika. DAP omogoča spomin 2900 procesorjem, hkrati pa mu je lahko ukazano s strani 2900 procesorjev, da izvrši določen DAP program. DAP je namenjen nekakšnemu "number crunching" procesiranju podatkov na vzporeden način.

Povezave med posameznimi komponentami DAP sistema, so prikazane na sliki 4.16. 4096 bitnoserijskih procesorjev je organiziranih v polje 64×64. Vsaka 64 bitna beseda v sistemu 2900 pripada vrstici DAP spomina.

Ortogonalna 64 bitna vrstična in stolpična povezava - vodilo, omogočajo dostop do poljubne vrstice ali stolpca spomina. Stolpična vodila prevzemajo podatkovne besede iz DAP spomina, prevzemajo vrstice ukazov kot povezuje kontrolno enoto, oziroma njene registre z DAP pomnilnikom. Ukazni



Slika 4.16: Arhitektura DAP sistema



Slika 4.17: Organizacija DAP večprocesorske enote

buffer lahko shrani do 60 ukazov. Vrstično vodilo pa prenaša podatke v obe smeri med registri glavne kontrolne enote in DAP pomnilnikom. DAP procesor je enostaven in ima veliko procesno moč.

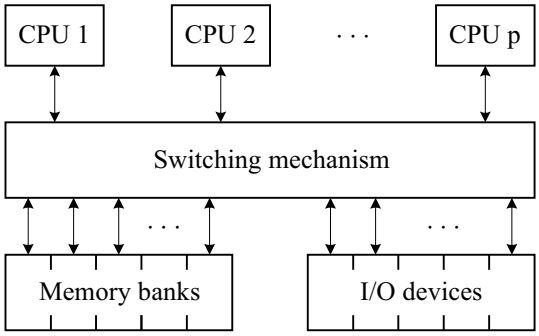
4.4 Multiprocesorji in multiračunalniki

MIMD računalnike sestavlja niz povsem programirljvih procesorjev, kjer lahko vsak izvaja lasten program. Multiprocesorje karakterizira porazdeljen - skupni pomnilnik. V nasprotju s tem ima vsaka CPE svoj spomin v takoimenovanih multiračunalnikih. Vse komunikacije med njimi se vrše z izmenjavo sporočil.

4.4.1 Tesno povezani multiprocesorji

Vsi tovrstni multiprocesorski sistemi delujejo tako, da posegajo po pomnilniku preko posebnega preklopnega mehanizma. Pri tem obstaja niz rešitev kako izvesti preklopni mehanizem.

Carnegie - Mellonov C.mmp in Sequentov Balance 8000 sta primera tesno povezanih multiprocesorjev.



Slika 4.18: Tesno povezan sistem

V primeru Balance 8000, kjer sistem uporablja vodilo, je nepraktično graditi velike sisteme zaradi hitrega nasičenja vodila z zahtevami posameznih procesorjev.

V primeru križnih stikal kot jih uporablja C.mmp, cena preklopnih stikal postane dominantni faktor glede na omejeno število procesorjev. Tesno povezani sistemi zasnovani na preklopnem omrežju lahko vsebujejo večje število procesorjev.

HEP je bil prvi komercialni tesno povezan multiprocesor zasnovan pri DENELCOR, Colorado. Zasnovan je na modulih. Vsak modul vsebuje 16 procesnih enot (PEM - process execution module) in 128 pomnilniških modulov povezanih s hitrimi preklopnimi vezji. Osnovna arhitektura HEP je razvidna s slike 4.19.

PEM je sposoben upravljati z večjim številom procesov hkrati. PEM je lahko izvedel 10^7 ukazov na sekundo (MIPS).

4.4.2 Šibko povezani multiprocesorji

Za šibko povezane multiprocesorje je karakteristično, da imajo porazdeljen naslovni prostor. Ta porazdeljen naslovni prostor je kombiniran z lokalnim spominom posameznih CPE. Čas dostopa do spominske lokacije je torej odvisen od lokacije pomnilniškega naslova, saj je ta lahko v okviru spomina klicujočega procesorja ali ne.

Cm*

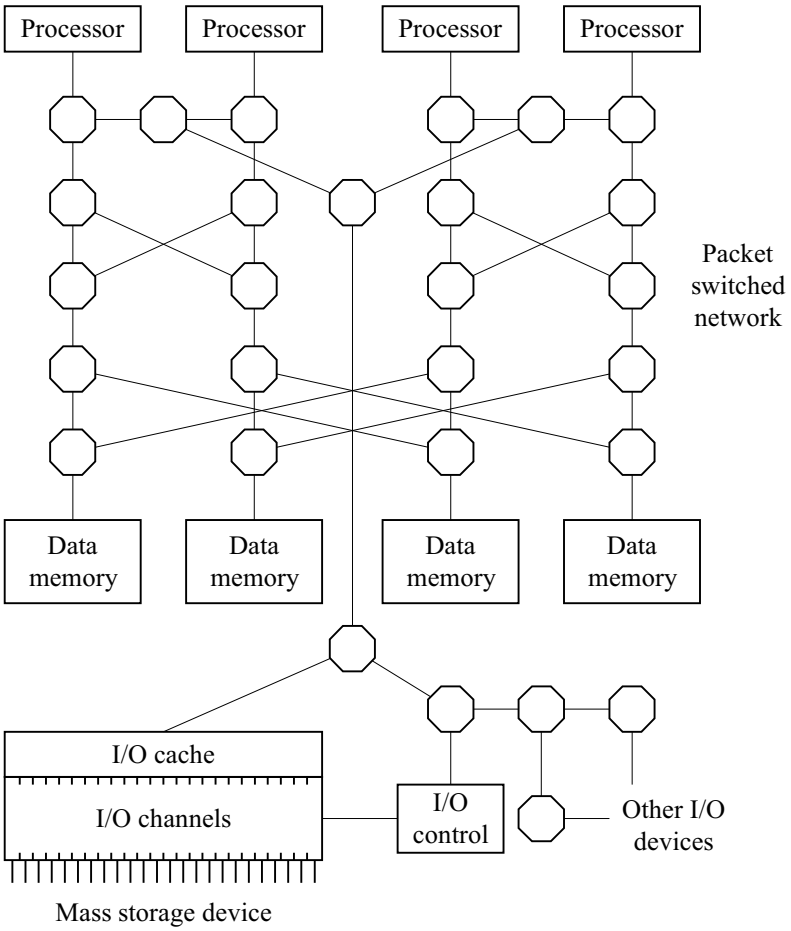
Na Carnegie Mellou University so razvili enega boljših šibko povezanih multiprocesorjev kot enoten računalniški sistem. Zasnovan je na modulih samostojnih računalnikov z lokalnim spominom in V/I enotami z ustreznim lokalnim preklopnim modulom. Nelokalna naslavljanja so usmerjena na "Map-bus" preko lokalnega stikala. Več računalniških modulov tvori grozd na enojnem "map-bus" vodilu (slika 4.21).

Grozdenje dovoljuje izmenjavo podatkov med procesorji. Zaradi omejitve prenosne kapacitete vodila je število računalniških modulov obešenih na "map-bus" omejeno. Zato se grozde povezuje preko medgrozdnih vodil (KMAP) v večji sistem. Če K-map kontroler zazna naslov, ki pripada drugemu grozdu, tedaj se naslov prenese k drugi K-map kontrolni enoti, ki prenese referenčni naslov npr. na svoje "Map" vodilo.

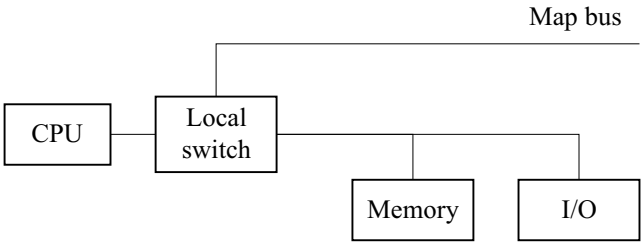
Tako v tem sistemu ni centralnega preklopnega mehanizma, kar omogoča priključitev večjega števila procesorjev. Cm vsebuje okoli 50 aktivnih procesorjev.

Newmanov metuljčni vzporedni procesor

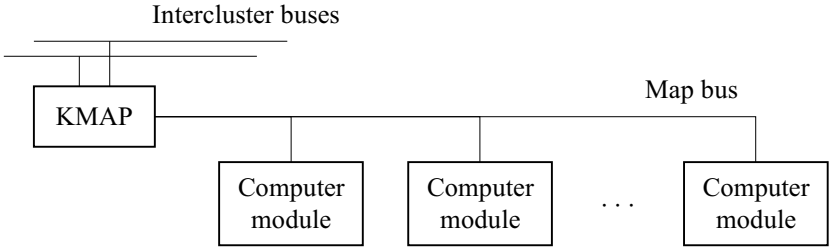
Ta sistem vsebuje od 1 do 256 procesorskih vozlov. V vsakem vozlu se nahaja EN z 1-4 Mbytov lokalnega spomina, koprocesor, ki deluje kot kontroler vozla (PNC), V/I vodilo in vmesnik za preklopno



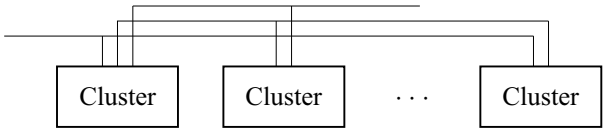
Slika 4.19: Arhitektura sistema HEP



Slika 4.20: Cm šibko povezani sistem

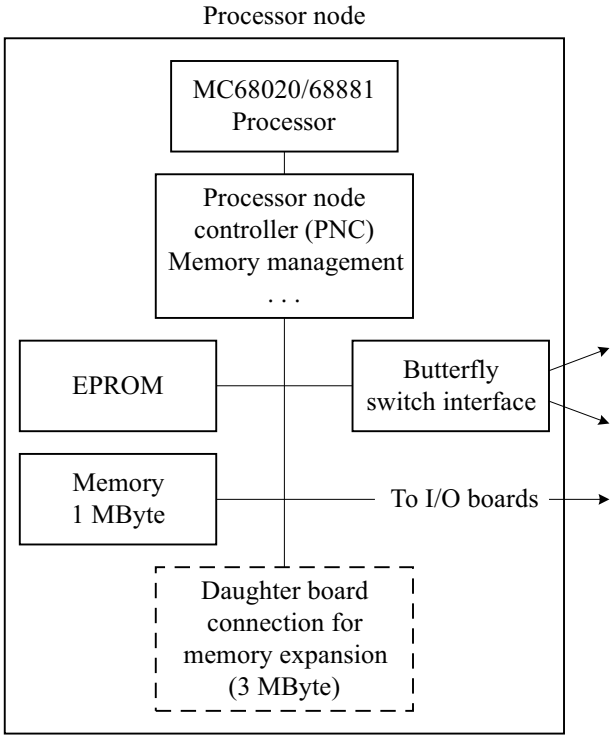


Slika 4.21: Povezave znotraj grozda



Slika 4.22: Medsebojno povezovanje grozdov

vezje.



Slika 4.23: Metuljčni vzporedni procesor

PNC omogoča transformacijo virtualnega naslova v realni naslov. Kot deluje PNC direktno z realnimi naslovi na lokalnem pomnilniku, tako lahko deluje tudi v okviru oddaljenih vozlov z uporabo virtualnih naslovov. Povezave med vozli so izvedene z uporabo metuljčne organizacije stikal.

4.4.3 Multi računalniki

Drug model MIMD računalnikov nima globalno porazdeljenega spomina. Vsak procesor ima svoj spomin. Izmenjava med procesorji se vrši na nivoju sporočil z uporabo spomina, razdeljenega med posamezne procesorje.

INTEL iPSC

To je standardni komercialni Intelov računalnik. Največji model sestavlja 128 mikroračunalniško zasnovanih vozlov povezanih v binarno hiperkocko (7-cube - sedem dimenzionalna kocka). Vsako vozlišče je povezano še s sedmimi ostalimi vozlišči preko bitne dvosmerne asinhrone povezave (100 Mb/s). Vsak vozel vsebuje kopijo jedra operacijskega sistema, tako da lahko izvaja in razvršča procese znotraj vozla . Prav tako ima vgrajene sistemske klice, ki omogočajo posameznim procesom, da medsebojno

komunicirajo in usmerjajo sporočila skozi vozlišče. Uporabnik komunicira s hiper kocko preko "cube managerja", ki lahko komunicira z vsakim vozlom posebej preko Ethernet kanala.

4.5 Lastnosti povezovalnih mrež

Povezovalne mreže so lahko statične ali dinamične. Statična omrežja omogočajo direktno povezavo točka-točka, pri čemer se struktura povezave ne spreminja v fazi izvajanja programa. Dinamična omrežja so dinamično rekonfigurabilna tako, da se čim bolje prilagodijo zahtevam uporabnikov. Tovrstna omrežja so sestavljena iz preklopnih kanalov, stikal in večstanjskih omrežij. Tu se najčešče uporabljajo sistemi s porazdeljenim pomnilnikom.

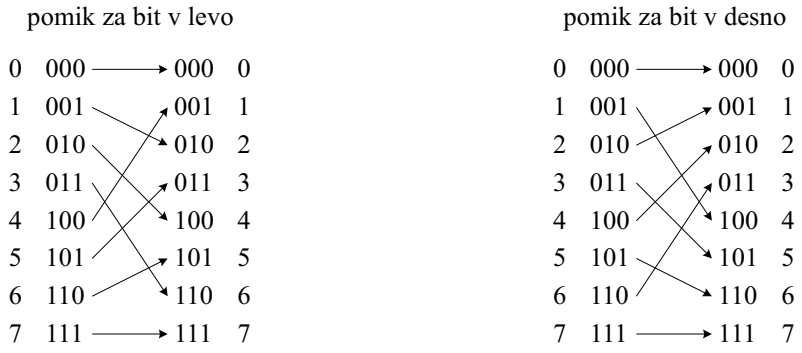
Obe vrsti omrežij pa lahko predstavimo z grafom, ki ima določeno število vozlov povezanih med seboj z usmerjenimi ali neusmerjenimi kanalnimi povezavami (linijami). Število vozlov v grafu določa dimenzije omrežja. Oglejmo si še nekaj parametrov, pomembnih za opis omrežja:

Red vozlišča Število povezav (kanalov) povezanih v en vozel določa red vozla d (node degree). V primeru usmerjenih povezav govorimo o usmerjenem vhodnem ali izhodnem redu vozla.

Premer omrežja D Je najkrajša pot med najbolj oddaljenima vozloma, ki se meri po številu potrebnih povezav med dvema vozloma. S komunikacijskega vidika mora biti premer omrežja D čim manjši.

Širina preseka Če določeno omrežje presečemo na dve enaki polovici, tedaj minimalno število presekanih poti predstavlja širino preseka b. V primeru komunikacijskega omrežja vsaka pot predstavlja kanal z W bitnimi povezavami. V tem primeru je širina presekane žične poti $B = b \times W$. Parameter B v tem primeru odraža povezovalno gostoto omrežja. V primeru, ko je B konstanten, je širina kanala določena v bitih $W = B/b$. Tako predstavlja širina preseka mero maksimalne širine komunikacijskega pasu v določenem omrežju.

Dolžina povezave ali kanala med dvema vozloma vpliva na zakasnitve, popačitve in zahtevano moč signala. Mreža je simetrična, če ima enako topologijo gledano iz vseh vozlišč mreže.



Slika 4.24: Metuljčno povezovalen ????

Povezovalno omrežje je namenjeno izmenjavi podatkov znotraj vozla oz. procesorskega elementa PE. Tovrstna omrežja so lahko zopet statična ali dinamična. Statična so npr. v hiperkocki TMC/CM-2, dinamična pa so pri IBM računalniku GF11. V multiračnalniškem omrežju se prevezovanje-povezovanje podatkov doseže z uporabo sporočil (message passing). Pri povezovanju sistemov s sporočili uporabljamo metode pomika (shifting), rotacije, permutacije ena-ena, razširjanja (broadcasting, multicasting), mešanja, zamenjave, itd. Tovrstne funkcije povezav uporabljamo na krogu, kocki ali večstanjskem omrežju (multistage network).

Permutacije: V primeru n objektov imamo $n!$ permutacij, po katerih razvrstimo te objekte. Običajno uporabljamo permutacijske skupine, kot npr. za permutacijo $\Pi = (a, b, c)(d, e)$. Tu imamo bijektivne preslikave $a \to b$, $b \to c$, $c \to a$ oz. $d \to e$ in $e \to d$. Prvi cikel (a, b, c) ima periodo 3 in drugi cikel (d, e) ima periodo 2. Če kombiniramo obe skupini Π, imamo periodo $2 \times 3 = 6$.

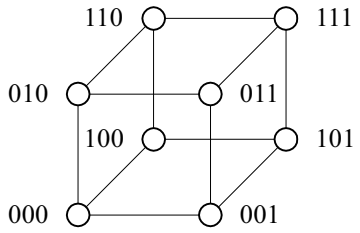
Permutacije se lahko uporabijo na "crossbar" stikalih ali na omrežjih delujočih s sporočili.

Popolno mešanje in zamenjava: Popolno mešanje je posebna permutacijska funkcija, uporabna pri vzporednem procesiranju. Preslikava v eno in drugo smer je vidna s slike a oz. s slike b.

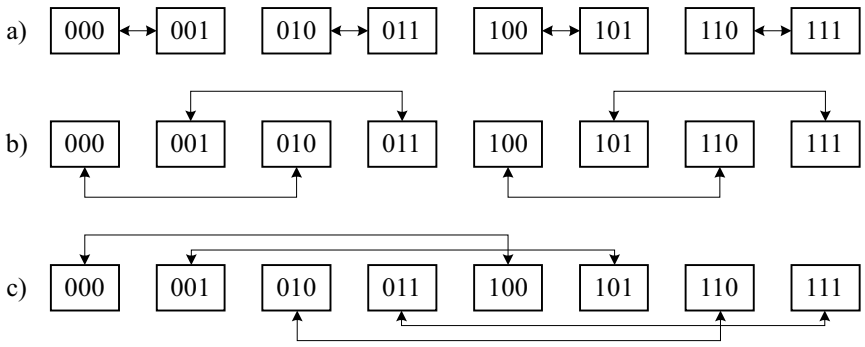
Pri popolnem mešanju lahko vsak objekt izrazimo s k–bitnim binarnim številom. $x = (x_{k-1}, x_{k-2}, \ldots, x_1, x_0)$. Popolno mešanje preslika x v y, kjer je $y = (x_{k-2}, \ldots, x_1.x_0, x_{k-1})$ s pomikom binarnega števila x za 1 mesto levo (ciklični pomik).

Povezovalna funkcija hiperkocke: Vzemimo trodimenzionalno binarno kocko in njene povezave — robove. Povezovalno funkcijo lahko določimo s tremi biti naslova vsakega vozla v kocki (C_2, C_1, C_0).

Primer: Vzemimo za primer povezave med vozli, ki se razlikujejo le za en bit na mestu LSB (C_0). Podobno kot v tem primeru (slika a), lahko določimo povezovalno funkcijo za srednji bit (C_1) - slika b in za bit z največjo utežjo MSB, bit C_2.



Slika 4.25: Povezave hiperkocke



Slika 4.26: Mešalno povezovanje

Razširjanje: (broadcast, multicast) Broadcast pomeni preslikavo iz enega vozla v vse ostale vozle (npr. v Array procesorju - nadzorna enota vsem ostalim PE polja). Multicast pa pomeni preslikavo iz ene podmnožice v drugo podmnožico.

Na lastnosti povezovalnega omrežja vplivajo naslednji parametri:

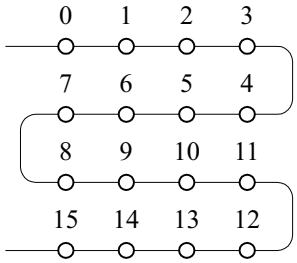
- funkcionalnost: pomeni, kako omrežje podpira povezovanje podatkov, kako delujejo prekinitve v omrežju, sinhronizacija, itd.
- zakasnitve omrežja: se odražajo v najslabšem primeru zakasnitve enotinega sporočila prenešenega po omrežju.
- pasovna širina: odraža maksimalen pretok podatkov po omrežju. Mb/s.

- materialna kompleksnost: glede na ožičenje, stikala, konektorje, arbitražo, vmesniško logiko, itd.
- modularnost: glede na zmožnost modularnega razširjanja in glede na linearno povečevanje lastnosti sistema.

4.6 Statična povezovalna omrežja

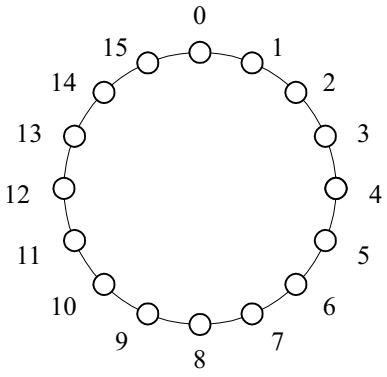
Statična povezovalna omrežja ostanejo fiksna, čim so zgrajena.

Linearno polje (Array) To je enodimenzionalno omrežje, kjer N vozlov povežemo z $(n-1)$ povezavami in predstavlja najenostavnejšo topologijo povezovanja. Pri velikem N je struktura povezav zelo neučinkovita. Za male N< 10 pa je tovrstna struktura tudi ekonomična.



Slika 4.27: Linearno polje ???

Obroč(obroč s tetivami) Tega dobimo s povezavo končnih vozlov linearnega polja. Obroč je lahko enosmeren ali dvosmeren. Obroč je simetričen, ko je red vozla $d = 2$. Premer omrežja D pa je N/2. Tipičen primer te topologije je IBM–ov obroč z žetonom (token ring). Paketno preklopljeni (povezani) obroč pa je uporabljen pri KSR-1 superračunalniku iz leta 1992.

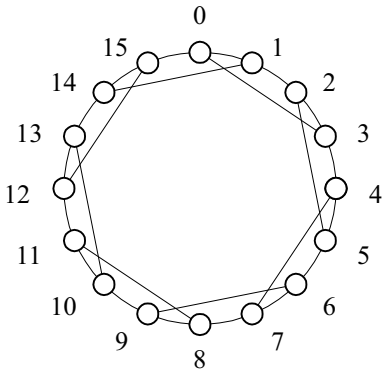


Slika 4.28: Obroč z redom voza $d = 2$ in premerom vozlišča D=8

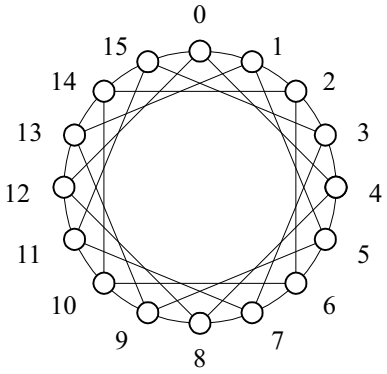
S povečevanjem reda vozla z 2 na 3 ali 4 dobimo obroče povezav z dodatnimi povezavami (s tetivami) (sliki 4.29).

V primeru s slike 4.29 z dvema tetivnima obročema premer omrežja D pade z 8 na 5, v naslednjem primeru z $d = 4$ pa pade D na 3. V ekstremnem primeru (primeru popolno povezanega obroča) pa pade premer omrežja na D=1.

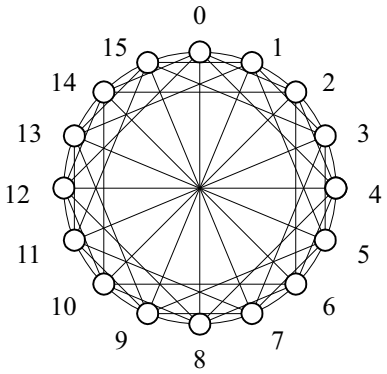
Pomični obroč (sod, Barrel Shifter) predstavlja obroč povezav, kjer vsak vozel i povežemo z naslednjim vozlom j tako, da je razdalja $|j - i| = 2^r$ za $r = 0, 1, 2, \ldots, n-1$ in kjer je dimenzija omrežja N=2^n. Red vozla v pomikalnem obroču je $d = 2n - 1$ in premer D=n/2. Za N=16, $d = 7$ in D=2.



Slika 4.29: Obroč z redom voza $d = 3$ in premerom vozlišča D=5



Slika 4.30: Obroč z redom voza $d = 4$ in premerom vozlišča D=3

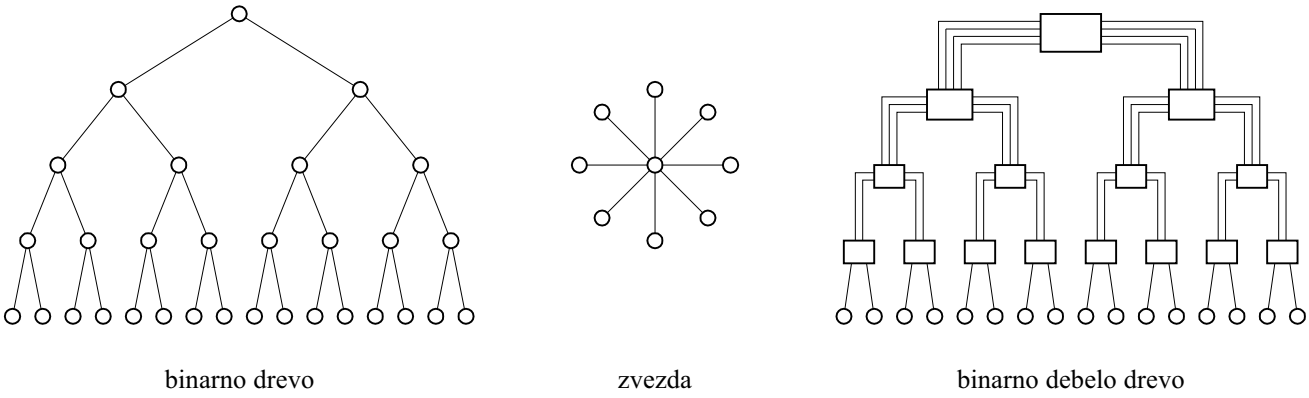


Slika 4.31: Pomični obroč

Drevesne in zvezdaste povezave: Na sliki je prikazano binarno drevo povezav z 31 vozli in petimi nivoji povezav. V splošnem ima tako binarno drevo na k-tem nivoju N=$2^k - 1$ vozlov. Maksimalna stopnja vozla je 3, s premerom D je $2(k - 1)$. Binarno drevo predstavlja modularno razširljivo arhitekturo.

Zvezdasta struktura s slike 4.32 je dvonivojsko drevo z redom vozla $N-1$ in konstantnim premerom 2.

Odebeljeno drevo: ima enako strukturo kot navadno drevo, le da je odebeljeno z dodatnimi povezavami. Leta 1985 je to strukturo povezav predlagal Leiserson. Pasovna širina povezovalnega kanala se povečuje, čim se premikamo od listov k (koreninam) deblu drevesa. S tem je odpravljeno

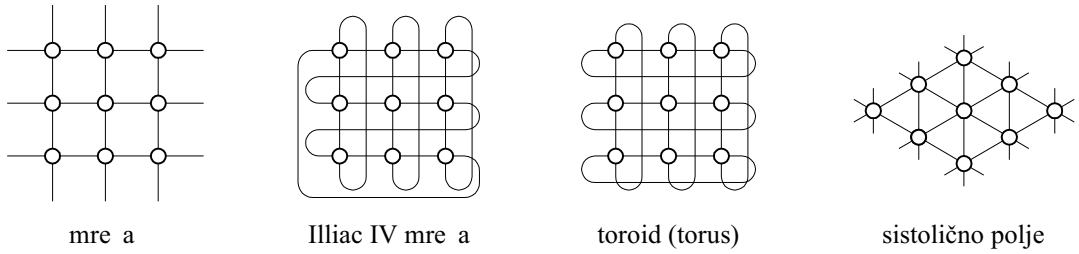


Slika 4.32: Drevesne in zvezdaste strukture povezav

ozko grlo povezovalnih kanalov pri klasičnem drevesu.

Ideja odebeljenega drevesa je bila uporabljeno pri CM-5 Connection Machine.

Mreže in Toroidi Mreža - povezovalna mreža je zelo razširjena v arhitekturi povezav in je uporabljena v ILLIAC IV računalniku, CM-2 Connection Machine, Intel-PARAGON superračunalniku itd.



Slika 4.33: Mrežne strukture povezav

V splošnem ima k-dimenzionalna mreža mreža z $N = n^k$ vozlov z notranjim redom vozla $2k$ in premerom omrežja $k(n-1)$. Navadna mreža dejansko ni simetrična, saj je red vozla na robu 3 oz. 2 na vogalu mreže.

Illiacova mreža je povezana tako, da so robovi med sabo povezani ali v isti vrsti ali z naslednjo vrstico. Mreža ILLIAC-a IV je ekvivalentna obroču z redom vozla 4 (vsak vozel ima 4 povezave) (slika 4.32 - obroč z vozlom $d = 4$).

Toroid (torus) ima krožno zaključene povezave vzdolž vsake vrste in vzdolž vsakega stolpca. Običajno je red vozla pri $n \times n$ binarnem toroidu $d = 4$ in premer je D=n. Toroid ima simetrično topologijo.

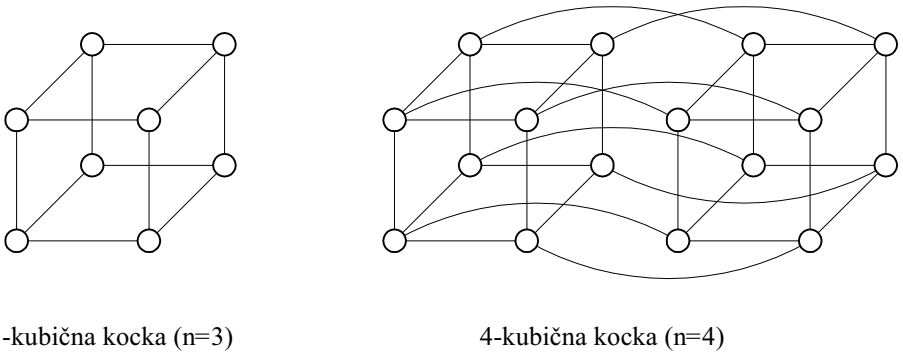
Sistolična povezovalna polja Ta spadajo v razred cevastih arhitektur, namenjenih predvsem reševanju konstantnih problemov. V primeru s slike je red vozla 6. Klasično gledano so sistolična polja cevaste strukture z mnogosmernim pretokom podatkov. Na tej osnovi je bil zgrajen superračunalnik INTELiWarp. Tovrstne strukture sta močno popularizirala ??? in Leiserson leta 1978.

V posebnih primerih, kot so problemi obdelave signalov, pa predstavlja tovrstna rešitev cenejšo in učinkovitejšo arhitekturo.

Polja imajo največkrat fiksne povezave, delujejo sinhrono v sami arhitekturi povezav in strukturi celic pa je vgrajen algoritem, ki ga rešujejo.

Hiperkocka To je binarna n-kubična arhitektura, ki je uporabljena v sistemih iPSC, nCUBE, CM-2. V splošnem govorimo o $N = 2^n$ vozliščih.

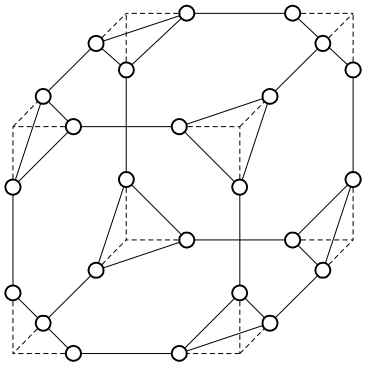
4-kubična kocka je sestavljena iz dveh trodimenzionalnih (3-kubičnih) kock, kjer so vozli obeh kock med sabo povezani, kot je prikazano na sliki 4.34.



Slika 4.34: Primera povezav kocke

Binarna hiperkocka je zelo popularna predvsem med raziskovalci. iIPSC/1 in iIPSC/2 in nCUBE so grajeni na osnovi hiperkocke. Hiperkocka vsebuje vse povezave kot so binarno drevo, mreže, itd. Stopnja razširljivosti in s tem modularnosti je majhna, saj je hiperkocka težko razširljiva, zato se jo nadomešča z drugimi arhitekturami.

Kockasto povezani obroči (CCC - Cube Connected Cycles) To je modifikacija hiperkocke. Dejanska ideja leži v povezanih vogalih (vozlih) kocke, kjer se ta nadomesti z obročem s tremi procesorji (slika 4.35).



Slika 4.35: Hiperkocka CCC

Na splošno lahko sestavimo k–povezovalne obroče v k–kocki z $n = 2^k$ obroči vozlov. Torej vsak vogal kocke nadomestimo z obročem s k vozli. k–kocka je torej sistem k-CCC z $k \times 2^k$ vozli. Premer povezovalne mreže je $2k$. Pri enostavni kocki je $k = 1$ in $2k = 2$.

V primeru 64–vozelnega CCC, se vogalna vozlišča nadomeste s 4 vozli.

k–polja n–kockastega omrežja

Poglavje 5

Paralelizmi

5.1 Paralelizmi v algoritmih

Kot smo v uvodu že omenili, je možno najti v večini algoritmov določeno vzporednost. Običajno posamezne probleme razdelimo na grobe podprobleme na nivoju aritmetičnih operacij, kar pomeni veliko problemov s sinhronizacijo v primerjavi s pridobitvami paralelizacije, ali pa ostanemo pri velikih jedrih, ki jih obdelujemo vzporedno. V primeru, ko se spustimo na nivo aritmetičnih operacij, naletimo na takoimenovano sinhrono vzporednost, v primeru velikih podproblemov, pa največkrat naletimo na takoimenovano asinhrono vzporednost.

V klasičnem primeru asinhrone vzporednosti govorimo o porazdelitvi podproblemov na posamezne procesorje. V slednje uvrščamo MIMD strukture, v sisteme z vgrajeno sinhrono vzporednostjo pa SIMD strukture.

5.1.1 Sinhronizacijski postopki in komuniciranje v MIMD sistemih

V poglavju o multiprocesorskih strukturah smo spoznali osnovne razlike med dvema MIMD strukturama in sicer, tiste z lokalnim in tiste z globalnim pomnilnikom. Kasneje je bil tudi uveden koncept virtualnega skupnega pomnilnika.

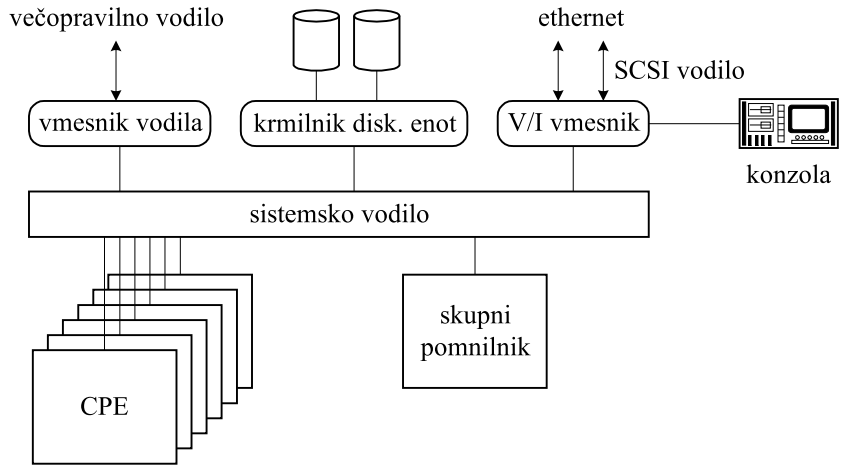
Procesorji delujejo v teh strukturah popolnoma samostojno, zato se morajo tudi sinhronizirati, če žele izmenjati določene informacije. Zato celotna programska struktura vsebuje samostojnost posameznih procesorjev.

Program običajno razdelimo na posamezne procese, ki tečejo asinhrono, a vzporedno. Idealna preslikava posameznih procesov je takšna, da vsak proces preslikamo na svoj procesor, kar pa v praksi največkrat ni mogoče. Današnji najčešči pristop je preslikava n procesov na 1 procesor. To pa pomeni, da mora procesor izvajati posamezne procese v določenih časovnih intervalih - časovno porazdeljeno.

Tipični predstavniki MIMD razreda vzporednih računalnikov so "Sequent Symetry", "Intel iPSC Hypercube" in "Intel Paragon". Procesorji, ki so povezani preko skupnega vodila so največkrat Intel 80586, 80486 (slika 5.1).

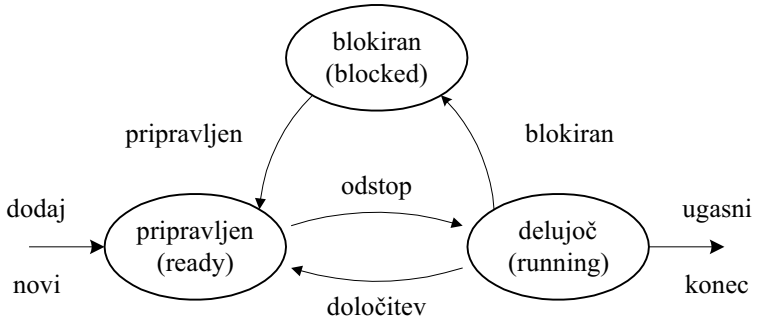
5.1.2 Stanja procesov

Proces predstavlja posamezen del programa, ki se izvršuje asinhrono, a neodvisno od ostalimh procesov. Ker se v splošnem na enem procesorju izvaja tudi več procesov, te procesor izvaja v določenih časovnih (rezinah) intervalih. Na koncu vsakega časovnega intervala, mora trenutno izvajani proces preiti iz stanja DELOVANJE (RUN) v stanje PRIPRAVLJEN (READY), v katerem čaka do naslednje, njemu pripadajoče časovne rezine. Zato se morajo vsi kontrolni podatki procesa shraniti v njegov "Process Control Block" (PCB). Novi procesi se takoj postavijo v vrsto READY - PRIPRAVLJEN, dočim se zaključni procesi odstranijo iz stanja DELOVANJA (RUNNING), s čimer sproste posamezne



Slika 5.1: Večprocesorski sistem

procesorje, za čakajoče procese. Procesi vedno čakajo toliko časa, dokler se ne izpolnijo določeni pogoji, kot so dostopnost do sistemskih virov ali sinhronizacija z drugimi procesi. Enako velja za blokirane procese, ki morajo odpraviti pogoje za blokado. Šele tedaj se jih razvrsti v stanje PRIPRAVLJEN. V trenutku, ko bodo izpolnjeni pogoji za stanje DELUJOČ, pa bo tak proces prešel v stanje DELUJOČ (slika 5.2).



Slika 5.2: Stanje procesov

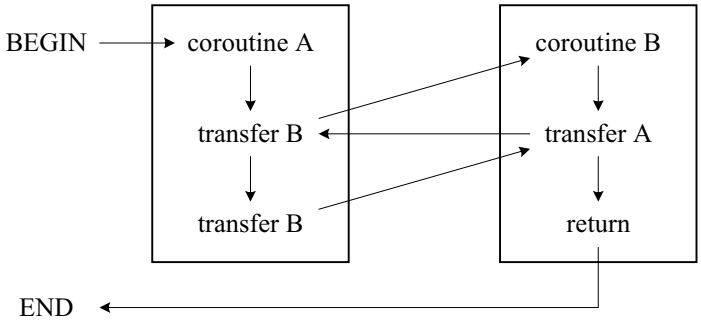
Del operacijskega sistema, ki skrbi za vse omenjene postopke na posameznih procesih, se imenuje razvrščevalnik (scheduler). Vsak procesor v sistemu (slika 5.1) vsebuje tak razvrščevalnik, ki nadzira izvrševanje posameznih sekvenc procesov nekega procesorja. V kasnejših poglavjih bomo opisali kompleksnejše postopke razvrščanja procesov "load balancing" zato, da bomo dosežegli enakomernejšo obremenitev posameznih procesorjev.

5.1.3 Osnovni pojmi vzporednega procesiranja

S temi pojmi se bomo srečevali v poglavjih o sinhronizacijskih postopkih.

Sorutine

Sorutine so posebna omejena oblika vzporednosti, ki jih zasledimo v jeziku Modula-2 (Wirth 83). Osnovni pristop velja za enoprocesorski sistem. Vzemimo samo ukazni niz (del programa) s sekvenčnim izvajanjem. Posamezni procesi so lahko zasedeni ali sproščeni, pri čemer uporabljamo sorutine na enostaven - direkten način.



Slika 5.3: Oblika vzporednosti z uporabo sorutin

S slike 5.3 vidimo nekakšno navidezno vzporednost, med katero so razporejeni soprogrami. Soprogrami so neke vrste postopki, kjer se ohranjajo lokalni podatki med klici sorutin. Izvajanje se vedno prične z nekim določenim soprogramom. Vsaka sorutina lahko vsebuje poljubno število prehodov, ki prenašajo nadzor iz ene na drugo sorutino. Ta prehod ni ekvivalenten klicu "procedure call". To pa pomeni, da v sorutini ni povratnega nadzora mesta klicajoče sorutine, ampak se izvrši enostaven preklop na drugo sorutino. V primeru, da že predhodno aktivna sorutina prevzame nadzor nad izvajanjem procesa, se izvajanje razume kot enostavno izvajanje prehoda. V primeru, ko se aktivna sorutina zaključi, se izvrši zaključek vseh sorutin. Prenos nadzora v tem primeru eksplicitno določi programer, ki mora paziti tako na pravilnost dostopa do sorutine, kot tudi na pravilno točko oziroma naslov prenosa nadzora v sorutini. Vejenje-delitev poteka nadzora je eksplicitno določeno s strani programerja in ker je bil ta koncept razvit za enoprocesorske sisteme, zato nikoli ne pride do prave vzporednosti izvajanja procesov.

Loči - združi (FORK-JOIN)

"Fork-join" konstrukt, znan v UNIX-u kot "fork" in "wait", sta vpeljala Conway in Van Horn in predstavlja enega prvih vzporednih konstruktov v vzporednih jezikih. V Unixu lahko startamo vzporeden proces z operacijo "fork". Na koncu čakamo, da se vsi procesi končajo z operacijo "wait". Čeprav je ta operacija prikaz prave vzporednosti, jo lahko še vedno izvajamo na enoprocesorskem sistemu, zasnovanem na časovnih rezinah.

V tem primeru vzporednega procesiranja se mešata dva, v osnovi različna koncepta:

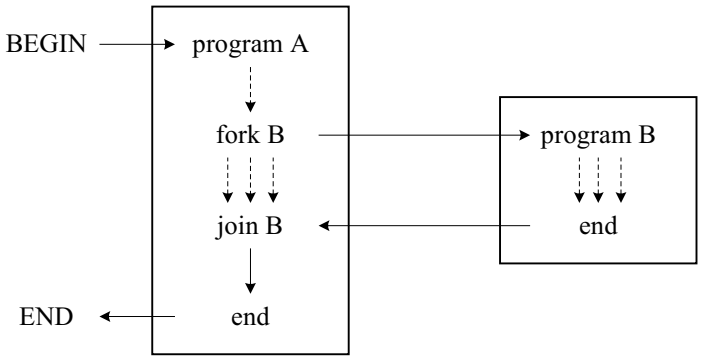
- deklaracija vzporednih procesov in
- sinhronizacija procesov.

Ker imamo opraviti s povsem različnima konceptoma, ju je smiselno ločiti v programskem jeziku na dve različni strukturi.

V Unix-u "fork" operacija, ki je predstavljena na sliki 5.4, ni tako splošna. Namesto skoka, se izvrši kopija klicočega procesa, ki se nato izvede vzporedno glede na originalni proces. Vrednosti vseh spremenljivk kot tudi programski števec so v novem procesu identične vrednostim v originalnem procesu. Edina možnost, po kateri ločimo procesa, je njegova identiteta (starši ali otrok). To pomeni, da lahko beremo identifikacijsko številko procesa, ki jo le-ta vrne pri operaciji "fork". Za proces otrok, je to število 0, dočim imajo vsi starševski procesi enako številko, kot je številka procesa v Unix-u, ki pa ni nikoli 0.

Zato, da pričnemo z novim programom in počakamo na njegovo izvršitev, sta bila v jezik C uvrščena dva klica Unix:

Klic operacije "fork" povrne številko procesa otroka starševskemu procesu (v tem primeru fork ni enak 0); seveda fork poda 0 procesu otroka. Na ta način proces otroka takoj izvede operacijo "execlp",



Slika 5.4: Operacija "loči-združi"

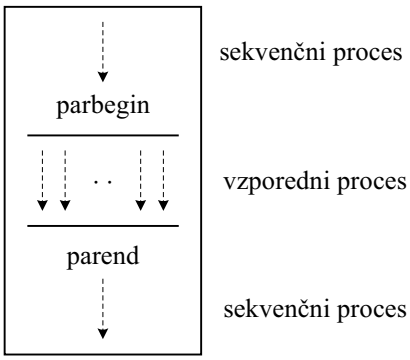
```
int status;
if(fork()==0) execlp (‘‘program_B’’,...);   /* Child Proc */
...                                          /* Child Proc */
wait (& status)
```

medtem pa proces staršev izvrši ukaz, ki sledi ukazu fork vzporedno s procesom otroka. Proces staršev nato čaka na zaključek procesa otroka in sicer z operacijo "wait". Povratni parameter vsebuje stanje procesa otroka.

Par Begin in Par End

Kot običajno pričnemo posamezne programske bloke z ukazom "begin" in jih zaključimo z ukazom "end", analogno v vzporednem programiranju pa pričnemo bloke s vzporedno kodo "parbegin" oziroma zaključimo s kodo "parend". Ukazi v bloku se izvajajo simultano. Tovrstni koncept je vsebovan v jeziku AL. Vzporedni program, napisan v AL lahko na primer nadzira več telefonskih klicev in koordinira njihovo izvajanje z uporabo semaforjev ali monitorjev. Tovrstna operacija se imenuje "cobegin" oziroma "coend".

Sinhronizacija z uporabo semaforjev je relativno enostavna in včasih nepotrebna. Razen omenjenih dveh konceptov ni višjenivojskih programskih principov sinhronizacije oziroma komunikacije, ki bi podpirali vzporedno programiranje.



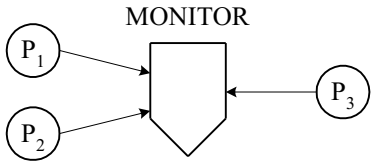
Slika 5.5: Par Begin Par End

Procesi

Procesi so, lahko rečemo, podobni proceduram - postopkom. Startamo jih z uporabo posebnega ukaza. V primeru, ko želimo izvesti hkratno nekaj kopij nekega procesa, moramo tako zvrst procesa pričeti z večkratnimi klici (multiple calls), ki naj bi vsebovali različne parametre. Sinhronizacija med hkratnim izvajanjem večih procesov se izvaja z uporabo semaforjev ali monitorjev in sicer z raznimi pogojnimi spremenljivkami (Hoare 74). Glede na semaforje, so monitorji zanesljivejši, prav tako pa nudijo višji nivo abstrakcije. Vsi porazdeljeni podatki so vkleščeni skupaj s podatki dostopa k operaciji, s čakalnimi listami v eno enoto imenovano "monitor". V monitorju se lahko v določenem času izvaja le en sam proces, kar zmanjšuje zahtevnost sinhronizacije. Blokiranje (blocking) in sprostitev (release) procesov znotraj monitorja se vrši z uporabo eksplicitnih operacij na pogojih čakalne liste.

Ta dodatna sinhronizacija vzporednih algoritmov prinaša v sistem dodatno obremenitev, hkrati pa predstavlja dodaten vir napak. V mnogih procesih potrebujemo dostop do porazdeljenega pomnilnika, ta skupna "kritična področja" pa moramo zaščititi s posebnimi sinhronizacijskimi pristopi. To pomeni, da je le enemu procesu v danem trenutku dovoljen pristop do tega kritičnega področja, kjer lahko obdeluje dovoljene mu podatke. Običajne napake se kažejo v nenadzorovanem dostopu ali izstopu iz tega kritičnega področja. Tovrstne napake vodijo do nekonsistentnih podatkov ali do smrtnega objema (deadlock) ali pa celo do blokade nekaterih procesorjev, v skrajnem primeru pa do blokade celotnega sistema.

Komunikacija in sinhronizacioja sta vgrajeni v sistem s porazdeljenim pomnilnikom (tesno povezani sistem) kot monitorja s pogoji. V sistemih brez porazdeljenega pomnilnika (šibko povezani sistem) nujno potrebujemo določene koncepte sprejema in oddaje sporočil med procesi oziroma procesorji.



Slika 5.6: Slika

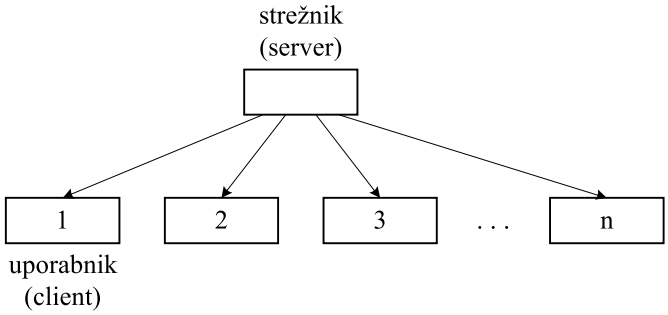
Oddaljeni klic postopka (Remote Procedure Call)

Če želimo razširiti koncept sinhronizacije na vzporeden računalniški sistem brez porazdeljenega pomnilnika, tedaj moramo zgraditi mehanizme za komunikacijo med procesi, ki se nahajajo na različnih procesorjih, z uporabo sporočil.

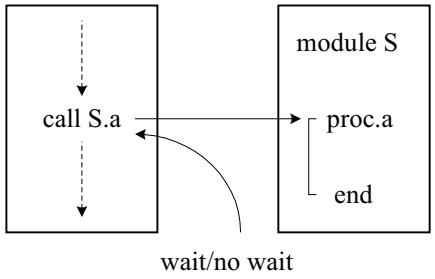
Programski sistem je razdeljen na več vzporednih procesov, kjer so nekateri procesi ali odjemalci ali strežniki (serverji). Vsak strežnik je v principu sestavljen iz neskončne zanke, kjer čaka na zahteve po posameznih računskih operacijah, oziroma vrača rezultat uporabniku. Vsak strežnik lahko tudi postane uporabnikc (client) z uporabo storitev drugega strežnika.

Tak način vzporedne porazdelitve nalog, je uporabljen v mehanizmu "Remote Procedure Call" (RPC). Procesna moč se s tem poveča, če seveda uporabnik nadaljuje s vzporednim procesiranjem, namesto da čaka na posamezne rezultate z nivoja strežnikov. Seveda pa s tem, ko želimo povečati izkoriščenost vzporedne materialne opreme, nastopijo tudi drugi problemi. Na primer povratni parametri posameznih strežnikov niso vedno na razpolago, kljub izvršitvi operacije na nivoju strežnika. "Remote procedure call" opravi le "deposit task" operacijo. Po izvršitvi zadane naloge in po vrnitvi rezultata, strežnik zahteva dodatno izmenjavo podatkov in sicer v smeri od strežnika k uporabniku.

Problemi z Remote Procedure Call vsebujejo protokole neobčutljive na napako ali na resetiranje, oziroma ponoven zagon odjemalca, v primeru napake pri strežniku.



Slika 5.7: Monitor in procesi



Slika 5.8: Primer oddaljenega klica

Vgrajena - implicitna vzporednost

Določeni vzporedni koncepti uporabljajo eksplicitne jezikovne konstrukte nadzora vzporednosti izvajanja procesa. Najidealnejši so tisti jeziki, ki ne vsebujejo posebnih konstruktov, pa vseeno omogočajo vzporedno procesiranje. Tovrstni programski jeziki se imenujejo jeziki z "implicitno vzporednostjo". Ti tudi predstavljajo najelegantnejši način vzporednega programiranja. Pri tem je programer dosti bolj omejen v možnosti nadzora izvajanja posameznih procesov na raznih procesorjih. Seveda moramo v tem primeru imeti opravka z inteligentnimi prevajalniki, ki pa morajo imeti dovolj podatkov v proceduri, če želimo doseči učinkovito paralelizacijo algoritma. Ta problem je dokaj jasen pri deklarativnih jezikih, kot sta LISP in PROLOG. Deklarativna predstavitev znanja o problemih, ki jih rešujemo, lahko jasno določijo rešitev. Seveda je težko pretvoriti to znanje v absolutni vzporedni program, to pomeni v generiranje programa, ki reši npr. enačbo tako, da porazdeli problem v vzporedno odvijajoče se podprobleme. Npr. matematična notacija seštevanja matrik vsebuje implicitno vzporednost in se v tem primeru dokaj enostavno prevede v ustrezno vzporedno računalniško arhitekturo, z uporabo postopkov avtomatske paralelizacije programa.

5.2 Sinhronizacijski postopki v sistemih MIMD

V sistemih MIMD je največkrat uporabljen sinhronizacijski postopek "Remote Procedure Call".

Pri izvajanju vzporednih algoritmov nastopita dva bistvena problema:

- zagotovitev nemotene izmenjave podatkov med procesi in
- preprečitev nekontroliranega dostopa večih procesov do iste spominske lokacije skupnega pomnilnika.

Ker se večino časa procesi izvršujejo neodvisno drug od drugega pomeni, da je izmenjava podatkov med njimi bistvenega pomena. Dva procesa, ki sodelujeta v izmenjavi podatkov, se izvajata asinhrono

in običajno na dveh različnih procesorjih. Pri izmenjavi podatkov se morata najprej sinhronizirati (uskladiti), ker do izmenjave podatkov lahko pride le, če sta oba procesa pripravljena. V primeru, če pride do izmenjave med procesorjema z uporabo porazdeljenega pomnilnika (to se dogaja, ko morajo procesorji izvesti več procesov), tedaj mora biti dostop zaporeden - sekvenčen. V drugačnem primeru lahko pride do napak v prenosu in npr. do smrtnega objema.

Običajno lahko vse sinhronizacijske operacije opišemo ali analiziramo z uporabo mrež Petri. /L/

Primer sinhronizacije dveh procesov si lahko predstavimo z dvema vlakoma, ki vozita po krožnih tirih, del tirnic pa je skupen. Vemo, da se na kritičnem - skupnem delu poti, lahko nahaja le en sam vlak. Sinhronizacijski proces lahko ponazorimo s semaforji. Pri tem lahko uporabimo niz različnih rešitev.

5.2.1 Uporaba sinhronizacijskih spremenljivk

Rešitev s pomočjo programske opreme

Peterson in Silberschaty kot tudi Ben-Ari opisujejo problem sinhronizacije preko porazdeljenega pomnilnika, ki ne potrebuje dodatne materialne opreme.

Vzporedna rešitev ne predstavlja nikakršnega problema za eno spremenljivko, saj že sam sistem poskrbi preko nadzora dostopa na vodilo, da le eden od procesorjev zavzema vodilo. Torej prireditev ene same besede ali spremenljivke ne predstavlja v multiprocesorskem sistemu nikakršnih težav. Problem nastopi pri zaščiti področij, kjer se shranjujejo posamezni parametri ali skupni podatki pred raznimi drugimi procesi.

Če npr., dva vzporedna procesa želita imeti dostop do porazdeljenega skupnega pomnilnika, tedaj operacije na podatkih, kot sta npr. branje in pisanje, predstavljajo operacije s <kritičnimi področji>. Ukazi, ki niso vpleteni v problem sinhronizacije, predstavimo kot <druge ukaze>.

Vzemimo dva procesa (P_1, P_2), ki startata zaporedno v glavnem programu, izvajata pa se lahko vzporedno.

```
- - -
start (P1);
start (P2);
- - -
```

Pristop 1

V tem primeru uporabimo za sinhronizacijo tako imenovano sinhronizacijsko spremenljivko. Oba procesa P_1 in P_2 imata dostop do skupne spremenljivke "turn"; branje ali pisanje te spremenljivke razumemo kot nedeljivo - atomično operacijo.

```
                Deklaracija:    var turn:1..2;
                Inicializacija: turn:=1;
                <izvajanje vzporednih procesov>
          P1                                    P2
loop                                  loop
 while turn <> 1 do <*nothing*> end;   while turn <> 2 do <*nothing*> end;
  <critical section>                    <critical section>
 turn:=2;                              turn:=1;
  <other instructions>                  <other instructions>
end                                   end
```

Če analiziramo oba programa (P_1,P_2), vidimo, da prikazana rešitev omogoča, da ima v določenem trenutku le en od proces dostop do kritičnega področja, kjer pa je zelo pomembno dejstvo, da je s

tem vsiljen sistem izmenjujočega dostopa do kritične sekcije. Slabosti tega pristopa so očitne. Oba procesa čakata v zanki pred vstopom na kritično področje, vse dokler spremenljivka "turn" ne zavzame vrednosti, ki pripada določenemu procesu.

Funkcionalnost sinhronizacije je takšna, da za procesom P_1 obstaja kritična sekcija, ko proces P_2 pristopi h kritični sekciji, šele nato proces P_1 vstopi do kritične sekcije. Ta pristop ni optimalen.

Pristop 2

V tem primeru sinhronizacijsko spremenljivko zamenjamo z nizom elementov (npr. Boolove spremenljivke), kjer vsak vsebuje le dva parametra, eden za vsak proces. Pred vstopom v kritično področje vsak proces postavi njemu pripadajoči element v stanje "true" in po izstopu postavi isti element v stanje "false". Na ta način vsak proces označi svojo prisotnost na kritičnem področju. Pred vstopom v to področje vsak proces čaka v "čakalni" zanki, vse dokler drugi proces ne zapusti tega področja.

```
            Deklaracija:    var flag:  array[1..2] of BOOLEAN;
            Inicializacija: flag[1]:=false, flag[2]:=false;
                 P1                                    P2
 loop                                  loop
   while flag[2] do <*nothing*> end;     while flag[1] do <*nothing*> end;
   flag[1]:=true;                        flag[2]:=true;
     <critical section>                    <critical section>
   flag[1]:=false;                       flag[2]:=false;
     <other instructions>                  <other instructions>
 end                                   end
```

Kljub vhodnemu testiranju se lahko dogodi, da oba procesa vstopita ali se nahajata v kritični sekciji istočasno. Če oba procesa torej izvršita testa v zanki "while" sočasno, tedaj lahko oba izstopita iz zanke in nadaljujeta v kritični sekciji. To pa je primer, kateremu se moramo izogniti.

Primer 3

Zato testiranje samo ni dovolj. Zato, da preprečimo možnost blokade prikazane v Pristopu 2, mora vsak proces nastaviti zastavico pred prvim testom v zanki "while". Zato vpeljemo v kritično sekcijo funkciji "testiranja" in "čakanja" v zanki.

```
   Deklaracija:    var flag: array[1..2] of BOOLEAN;
   Inicializacija: flag[1]:=false, flag[2]:=false;
          P1                                      P2
 loop                                  loop
   flag[1]:=true;                        flag[2]:=true;
   while flag[2] do (*nothing*) end;     while flag[1] do (*nothing*) end;
   flag[1]:=false;                       flag[2]:=false;
     <critical section>                    <critical section>
   flag[1]:=false;                       flag[2]:=false;
     <other instructions>                  <other instructions>
 end.                                  end.
```

V primeru 3 je dostop do kritičnega področja zavarovan, zato je lahko na kritičnem področju le en sam proces. Če pa oba procesa hkrati postavita zastavici, tedaj pride do blokade, saj oba procesa čakata v zaprti zanki "while".

V takšnem pristopu smo dosegli zaželen pristop medsebojnega izključevanja. V primeru, da vsak proces postavi zastavico v stanje "true" in če potem oba čakata v njuni "while" zanki na drugi proces,

ki naj zaključi delo v kritičnem področju in ki naj resetira zastavico, potem se blokira celoten proces. Nihče od dveh procesov ne more izstopiti iz zanke in vzporedni sistem je tako blokiran.

Primer 4

Ta pristop je glede na dosedaj omenjene pristope najkompleksnejši, a je hkrati tudi zadovoljiv. Algoritem je prevzet po /Petersonu/ in je sličen Dekker-jevemu algoritmu. V tem primeru prevzamemo oba dosedanja pristopa in ju združimo. V algoritmu nastopa spremenljivka "turn" in niz "flag". Algoritem nastavi spremenljivko "turn" na številko procesa, ki naj bi se šele izvajal (drugi proces) in čaka vse dokler ima drugi proces zastavico v stanju "set" in "turn" vsebuje številko drugega procesa. Po izstopu iz kritičnega dela, vsak proces resetira svojo zastavico (spremenljivko).

```
   Deklaracija:    var turn: 1..2;
                   flag: array[1..2] of BOOLEAN
   Inicializacija: turn:=1;
                   flag[1]:=false; flag[2]:=false;
          P1                                        P2
 loop                                      loop
   flag[1]=true;                             flag[2]:=true;
   turn:=2;                                  turn:=1;
   while flag[2] and (turn=2)do              while flag[1] and(turn=1) do
     <*nothing*> end;                          <*nothing*> end;
     <critical section>                        <critical section>
   flag[1]:=false;                           flag[2]:=false;
     <other instructions>                      <other instructions>
 end.                                      end.
```

V primeru uporabe tega algoritma je proces dostopa do kritičnega področja varen pred več kot enim procesom hkrati. Prav tako se ne more pripetiti blokada procesov.

Poseben test v "while" zanki preprečuje blokado procesov. Do čakanja pride samo v primeru, ko drugi proces postavi svojo zastavico in se dejansko nahaja v kritičnem področju. Čim proces, ki si je prvi pridobil možnost dostopa do kritičnega področja, zapusti to področje, tedaj proces resetira svojo zastavico in drugi proces lahko zapusti čakalno zanko in vstopi v kritično področje.

Eisenberg in McGuire sta razvila opisan postopek za n procesov.

Rešitev s pomočjo strojne opreme

Kot smo videli, je sinhronizacija večih procesov v multiračunalniškem sistemu dokaj zahtevna . Zato se najčešče uporablja realizacijo nadzora sinhronizacije nad procesi z uporabo strojne opreme.

V enoprocesorskem sistemu, kjer sistem deluje v skladu z organizacijo časovnih rezin, lahko dosežemo sinhronizacijo z blokado ali dopustitvijo nastopa prekinitev. Zanimiv pristop je "test-and-set", ki se najčešče uporablja v multiprocesorskih sistemih. Ta predstavlja enostavno strojno rešitev in se izvede v dveh korakih:

- branje Boolove spremenljivke in
- postavitev te spremenljivke v "true".

```
procedure test_and_set
  (var lock: BOOLEAN): BOOLEAN;
   var mem: BOOLEAN;
       begin_atomic (*nelocljive operacije*)
          mem:=lock;
```

```
        lock:=true;
        return(mem)
    end_atomic; (*nelocljive operacije*)
```

Oba koraka morata biti izvedena drug za drugim kot nedeljiva operacija. Test-and-set je znana kot atomična operacija. V času njene aktivnosti nima nobena ostala enota dostopa do vodila ali do prekinitve. Z uporabo te operacije je sinhronizacija večih procesov dokaj enostavna. Resetiranje spremenljivke v stanje "false" ne zahteva nikakršnih dodatnih preverjanj.

Vsak proces čaka v "busy wait" zanki vse dokler sinhronizacijska spremenljivka "lock" ne preide v stanje "false". V fazi prvega uspešnega testa z uporabo "test-and-set", se takoj postavi v stanje "true". Tako proces takoj vstopi na kritično področje. Po izstopu iz tega področja resetira spremenljivko - postavi jo v stanje "false", nakar lahko naslednji proces dobi dostop do kritičneg področja z uporabo "test-and-set".

```
Deklaracija:    var lock: BOOLEAN;
Inicializacija: lock:=false;

                 Pi
loop
  while test_and_set(lock) do <*nothing*> end;
     <critical sector>
  lock:=false;
     <other instructions>
  end.
```

Zahtevnejši sinhronizacijski principi zamenjujejo čakalno zanko z učinkovitejšimi vrstami sinhronizacije.

5.2.2 Uporaba semaforjev

Semaforji so bili kot vsi drugi sinhronizacijski principi uvedeni zato, da omejimo dostop do kritičnih področij v sistemu in so učinkovitejši kot so predhodno obravnavane "busy wait" zanke.

Semaforje kot sinhronizacijsko metodo, je uvedel Dijkstra 1965. leta. V najenostavnejšem primeru lahko semafor zavzame le dve vrednosti: prost in zaseden. Operaciji za postavitev ali resetiranje signala semaforja sta P in V po nizozemskih besedah za "prešel skozi" in "zapustil".

Primer:

```
 Pi
----
P(semafor);
   <critical section>
V(semafor);
----
```

Konceptualno gledano je konstrukt semaforja enostaven, dejansko "okleščimo" kritični del programa s parom P, V. Kritično je le, da vsak proces, ki posega v dani del porazdeljenega pomnilnika uporablja iste spremenljivke semaforja. Torej semafor ne pripada posameznim procesom, temveč porazdeljenim podatkom.

Uporaba: Semaforji se prikazujejo kot zapis podatkovne strukture, v katero se zapisujejo vrednosti glede na vrstni red zahtev posameznih procesov. V najenostavnejšem primeru rečemo, da je semafor Boolova spremenljivka. V tem primeru govorimo o Boolovem semaforju. Po drugi strani pa lahko semafor zavzame celoštevilske vrednosti (tipa integer). Temu pa pravimo splošni semafor.

```
type Semaphore=record
  value: INTEGER;
  L:     List_of_ProcID;
  end;
var: s:Semaphore
```

Povsem jasno je, da sta operaciji P in V atomični (nedeljivi) nerazdružljivi operaciji, zato imajo večkratni procesi dostop do skupnih podatkov semaforja.

Inicializacija semaforja

Parametra:

S.L ← prazna lista
S.value ← število dovoljenih operacij P brez nastopa operacij V.

Medtem, ko z enim semaforjem (value ≡ 1) zaščitimo kritične predele, najdemo tudi primere, kjer so smiselne tudi višje vrednosti parametra "value". Če je vrednost večja kot ena, to pomeni, da je število operacij P, ki se lahko dogode brez vmesnih operacij V; in sicer pred tem, ko klicani proces postane blokiran s semaforjem. Če je proces blokiran s semaforjem, mora ta počakati na naslednjo operacijo V sledečega procesa.

```
procedure P (var s:semaphore);
begin
  s.value:=s.value-1;
    putlast(s.L,actproc); (*add this process to queue S.L*)
    block(actproc)        (*and put it in state 'blocked*)
  end;
end P;
```

Operacija P zmanjša vrednost semaforja za ena in testira ali je vrednost že manjša kot 0, kar pomeni, da so semafor in njemu pripadajoči viri zavzeti. V tem primeru se klicoči proces postavi kot zadnji v semaforjevo vrsto in se blokira.

```
procedure V (var s:Semaphore);
var Pnew: Proc ID;
begin
  s.value:= s.value+1;
  if s.value <= 0 then
    getfirst(S.L,Pnew); (*odstrani proces P iz vrste S.L *)
    ready(Pnew)         (*in ga postavi v stanje 'ready'*)
  end
end V;
```

Operacija V poveča vrednost semaforja za ena, nakar testira, ali je le-ta še vedno manjša ali enaka 0. Če je temu tako, potem se naslednji čakajoči proces odstrani iz semaforjeve vrste in se doda k vrsti procesov, ki so v stanju "ready".

Pri V operaciji se lahko branje in postavljanje vrednosti semaforja realizirata programsko ali s pomočjo ustrezne materialne opreme. Pri programski realizaciji bi lahko uporabili "busy-wait" zanko, ki pa je dejansko neučinkovita operacija. Kritična sekcija programa vsebuje le tri ali štiri elementarne operacije za P ali V operaciji. To pomeni, da je v primeru nastopa faze čakanja, to posledica majhnega števila osnovnih ukazov.

Rešitev z dodatno materialno opremo je združena z uporabo "test- and-set" ukaza in sicer pred izvedbo P ali V operacije. Celo v tem primeru ni mogoče preprečiti kratkih čakanj do katerih lahko pride.

Problem pa nastopi, če v času izvajanj štirih osnovnih operacij proces prekorači dolžino časovnega intervala (time slice). Ta problem postane zelo važen takrat, ko se tako prekoračanje zgodi na nivoju vrste in ne na nivoju čakalne zanke. Nekateri jeziki, kot je Modula-2 uporabljajo takšen način sinhronizacije procesov.

Oglejmo si nekaj primerov uporabe semaforjev.

Problem proizvajalec - uporabnik (Producer - Consumer)

Procesa, ki si izmenjujeta podatke preko skupnega pomnilnika, se morata sinhronizirati. Eden generira podatke in jih sestavlja v vmesni pomnilnik ("bufer"), dočim jih drugi prebere in nadalje obdeluje. Princip uporabe semaforjev preprečuje zapis novih podatkov preko že vpisanih a neprebranih podatkov, kot tudi večkratno branje istih podatkov. Do tega pride, ko je en proces, npr. proizvajalec hitrejši od drugega, npr. uporabnika.

V tem primeru uporabimo Boolov semafor, s katerim nadziramo dostop do področja enojnega "buferja". Eden informira, kdaj je "bufer" prazen, drugi pa informira uporabnika, da je "bufer" poln.

```
    Deklaracija in inicializacija:

    var empty: semaphore[1];
        full : semaphore[0];
```

```
process Proizvajalec;
begin
  loop
  <generiranje podatkov>
  P(empty);
    <napolni buffer>
  V(full);
  end;
end process Producer;
```

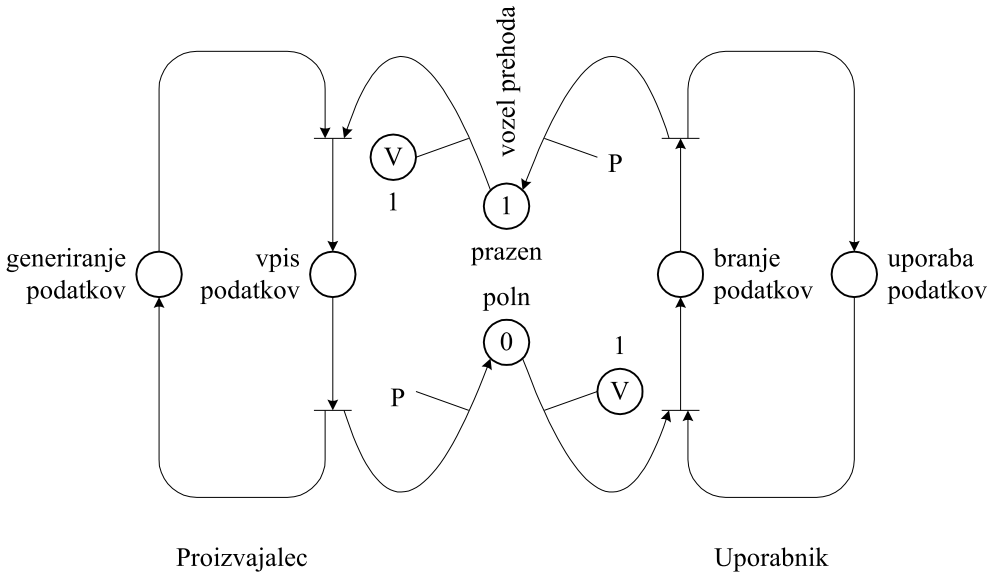
```
process Uporabnik;
begin
  loop
  P(full);
    <izprazni buffer>
  V(empty);
    <procesiraj podatke>
  end;
end process Consumer;
```

Predno startamo s procesom, postavimo semafor "empty" na vrednost 1, semafor "full" pa na 0, kar pomeni, da je "bufer" prazen. V tem primeru lahko proizvajalec takoj generira nove podatke in z operacijo P na semaforju testira ali je "bufer" prazen (empty). Če je prazen, tedaj proizvajalec napolni "bufer" in signalizira z operacijo V nad semaforjem (full), da je "bufer" poln in da je pripravljen za uporabnika. Sedaj uporabnik prebere podatke iz podatkovnega "buferja", ter izvede operacijo V na semaforju (semaphore empty). S tem signalizira proizvajalcu, da ponovno napolni vmesnik.

Mreža Petri prikazuje problem sinhronizacije. Vsak proces, proizvajalec in porabnik, opisuje cikel, ki je predstavljen kot neskončna zanka. Vsak cikel vsebuje dve stanji. Operacija P na enem od semaforjev "prazen" ali "poln" je predstavljena z vhodno povezavo k vozlu prehoda, dočim operacija V je predstavljena z izhodno povezavo iz vozla prehoda.

Bufer z omejitvijo

V predhodnem primeru sta proizvajalec kot uporabnik delovala več ali manj neodvisno. Pri tem se lahko zgodi, da je npr. uporaba podatkov dolgotrajnejša od generiranja. To pa pomeni, da se morata procesa čakati. Zaradi posameznih zakasnitev pride še do dodatnih zakasnitev v izvajanju celotnega programa. Zato vpeljemo večji vmesni pomnilnik, ki pa je še vedno omejen. V našem primeru naj le-ta vsebuje N pomnilniških elementov. Pristop zahteva Boolov semafor za zaščito kritičnih področij ter dva posplošena semaforja, ki ponazarjata koliko sta oba "buferja" zasedena.



Slika 5.9: Mreža Petri problema proizvajalec-uporabnik

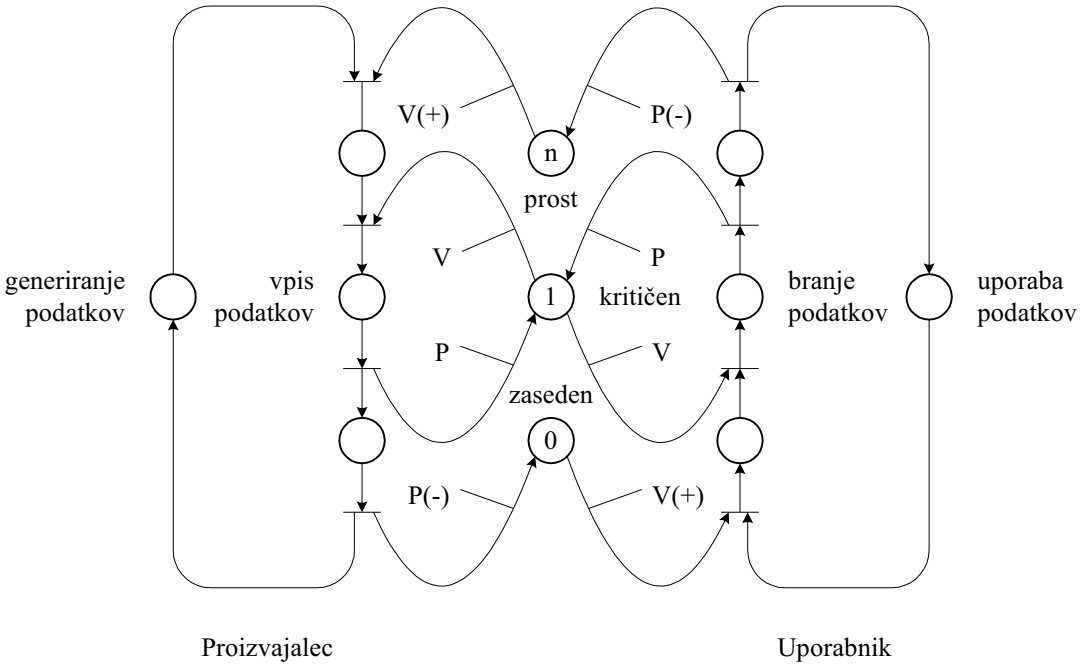
```
     Declaration and Initialization:
        var critical: semaphore[1];
            free    : semaphore[n];
            occupied: semaphore[0];
proces Producer;
begin
  loop
  <generiranje podatkov>
    P(free);
      P(critical);
        <vpisi podatke v buffer>
      V(critical);
    V(occupied);
  end;
end process Producer
```

```
proces Consumer;
begin
  loop
    P(occupied);
      P(critical);
        <beri podatke iz bufferja>
      V(critical);
    V(free);
      <procesiraj podatke>
  end;
end process Consumer;
```

V fazi inicializacije postavimo ustrezne parametre, npr. "occupied" vsebuje vrednost 0, kar pomeni, da je "bufer" prost za n elementov. Semafor critical je inicializiran na 1, kar pomeni, da dopuščamo vstop le enemu procesu na kritično mesto. Operaciji branja in pisanja procesov sta vsebovani v P in V operacijah na semaforju kritičnega dela. "Bufer" lahko hkrati vsebuje proste in zasedene elemente, dočim prejšnji princip - proizvajalec - porabnik - pa je vedno govoril le o enem samem elementu "buferja" (poln ali prazen).

Na isti način uporabnik čaka, da proizvajalec izvede operacijo P na semaforju "free", nakar ta prične z operacijo P na semaforju "occupied", prebere podatke in izvede operacijo V na istem semaforju "free". Sedaj se semantika malo spremeni, ker operiramo z večimi elementi v "buferju" z dvema posplošenima semaforjema. Števila "free" in "occupied" se povečujejo ali zmanjšujejo znotraj semaforjev "free" in "occupied", seveda glede na to katera operacija, V ali P je bila uporabljena.

Ta sinhronizacijski mehanizem je prikazan kot razširjena mreža Petri (slika 5.10). Glede na prejšnjo sliko smo Boolove semaforje razširili na splošne semaforje predstavljene z vozli stanj (krogi) s poljubnim številom žetonov. Dodatni Boolov semafor critical je potreben za sinhronizacijo večkratnih



Slika 5.10: Razširjena mreža Petri problema omejenega "bufferja"

proizvajalcev in porabnikov. Števec occupied predstavlja število trenutno zasedenih lokacij pomnilnika - "bufer", dočim free predstavlja še število prostih lokacij pomnilnika - "buferja".

Primer Beri - piši

V mnogih aplikacijah skupina procesov zahteva (ekskluzivni) dostop do določenih virov, dočim ostali procesi lahko dostopajo simultano.

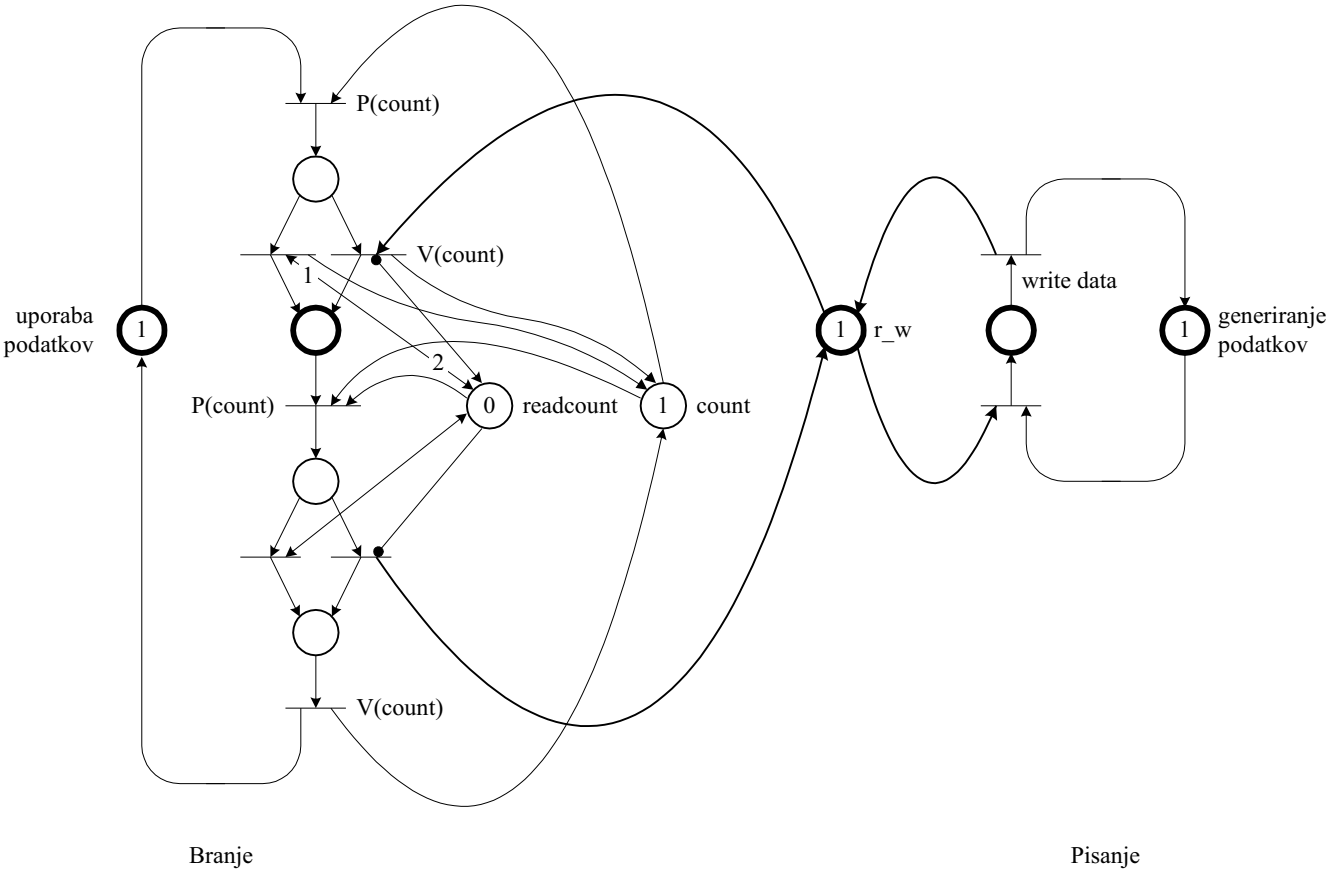
Najčešče se ta pojavlja pri dostopu do skupnega pomnilnika, ko nekateri procesi več vpisujejo, drugi pa več berejo. Kljub temu, da lahko več procesov hkrati zahteva dostop do pomnilnika, je le enemu izmed njih dovoljen dostop do tega pomnilnika in pisanje vanj. V času pisanja ni drugim procesom dovoljeno niti branje samo zato, da bi se preprečile napake pri vpisu. V primeru, ko uporabimo le en sam semafor, tedaj lahko dostopa do področja le en sam proces. To pa pomeni izgubo prednosti vzporednosti sistema. Rešitev je v problemu, ki so ga nakazali že Courtois, Hequas, Parnas, ki so uvedli princip maksimalnega možnega paralelizma, znanega kot "problem beri - piši".

```
Declaration  var count: semaphore[1]
                 r_w  : semaphore[1]  reading-writing
             readcount: INTEGER;
Initialization: readcount:=0;

process Reader;
begin
  loop
    P(count);                          proces Writer
      if readcount=0 then P(r_w)       begin
      end;                               loop
      readcount:=readcount+1;              <generate data>
    V(count);                              P(r_w);
```

```
      <read from data buffer>                <write data into buffer>
    P(count)                               V(r_w);
      readcount:=readcount-1;            end; (*loop*)
      if readcount=0 then V(r_w)       end process Writer;
      end;
    V(count)
    <process data>
  end; (*loop*)
end process Reader;
```

Vsak bralec mora preveriti, ali je on prvi proces, ki namerava pristopiti k branju skupnega pomnilnika. Drugi bralci - procesi povečujejo števec "readcount", vendar ne izvrše operacije nad semaforjem. Prvi vpisovalec - proces, ki bi rad pristopil k skupnemu pomnilniku, bo blokiran v klicu operacije semaforja P (r-w). Čim proces - bralec zapusti kritično področje, tedaj se števec "readcount" zmanjša za 1. Zadnji bralec izvede operacijo na V (r-w) semaforju. S tem klicem, je čakajočemu procesu - pisanja povrnjen - predan ekskluzivni dostop do spomina. Novi procesi bralci ali vpisovalci morajo počakati vse dotlej, dokler se ne konča ekskluzivni proces.

Operacija štetja kot pogojne operacije na semaforju, povzroči nastanek dveh novih kritičnih področij, ki jih moramo zaščititi z dodatnim semaforjem count.



Slika 5.11: Slika

S to rešitvijo se lahko dogodi pomanjkanje časa za proces pisanja pri dovolj velikih količinah bralnih procesov. S procesom branja konstantno prihajamo in zapuščamo kritično področje pomnilnika tako, da se lahko dogodi celo, da procesu pisanja ne bo nikoli dovoljen dostop. Temu se izognemo

s kompleksnejšo rešitvijo, v kateri procesu branja ni več dovoljen vstop na kritično področje čim se pojavi zahteva po generiranju procesa pisanja.

Slika 5.11 prikazuje mrežo Petri omenjenega problema. Zaradi enostavnosti sta bila uporabljena le en proces branja in le en proces pisanja. Povezave z osrednjim semaforjem (r-w) so odebeljene, povezave nazaj in naprej med istimi vozli stanj in vozli prehodov so označeni dvojno zaradi lažjega branja. V primeru večkratnih bralnih procesov, si ti delijo isti vozel (count) mreže Petri oziroma vozel (readcount) in vozel (r-w). Po zavzetju semaforja (count), kar pomeni po odvzemu žetona, s strani prvega procesa, se izvede izbira z uporabo inhibitornih - zaviralnih povezav. Ta selekcija je seveda odvisna od spremenljivke (vozel), readcount, ki je lahko nič ali večja kot nič. Samo tedaj, ko je (readcount) nič se sproži ustrezen vozel prehoda in s tem operacija P (r-w) - na osrednja semaforja (r-w), pri čemer se odstrani (zmanjša) vrednost žetonov za 1.

V obeh primerih sprožitev prehoda povzroči zmanjšanje vrednosti spremenljivke read-count za 1. Šele sedaj se lahko izvede proces branja na skupnem pomnilniku. Čim pa proces želi končati postopek branja skupnega podatka, mora najprej zaseči semafor count, šele nato lahko izvede odštevanje spremenljivke readcount za 1. Samo v primeru, ko je ta spremenljivka (readcount) nič (0), kar pomeni, da je bil to zadnji proces v vrsti bralnih procesov, se lahko izvede operacija V na semaforju r-w, ki pa povzroči sprostitev semaforja count (postavi žeton v count).

Postopek procesa vpisovanja je enostavnejši. Žeton semaforja r-w moramo odšteti pri vsakem vstopu preko vstopne povezave. Žeton se po vsakem posegu (zapisu) obnovi z uporabo izhodnih povezav. (Izvede se operacija V).

V sistemih podatkovnih baz uvedemo takoimenovani "porazdeljen" (shared) semafor. Tega lahko uporablja več procesov v "multiple" načinu delovanja ali en sam procesor v "exclusive" načinu delovanja. Porazdeljen semafor je poslošen model semaforja. Semaforji so enostavni sinhronizacijski mehanizmi, vendar se lahko ravno zaradi te enostavnosti vnaša v sistem veliko sinhronizacijskih napak. To se izraža tudi v tem, da kot uporabniki pozabimo na operacijo V ali P, ali celo na nepravilnosti pri delovanju s procesi. Zato lahko prihaja do napak kot so sprostitev semaforja in sicer v posebnih pogojih kompleksnejših procesov. V sistemih s takoimenovanimi funkcionalnim "recovery" postopkom, lahko predvidimo določeno mero tolerance do nastopa napak v sinhronizaciji. Ta pristop omogoča sprostitev semaforjev in zaključek procesov.

5.2.3 Uporaba monitorjev

Monitorje je uvedel Hoare leta 1974 kot razvitejšo in učinkovitejšo metodo sinhronizacije vzporednih algoritmov in predstavljajo abstrakten podatkovni tip.

Monitor vsebuje podatke, ki jih varuje, kot tudi ustrezen dostop do sinhronizacijskih mehanizmov. Dostop in sinhronizacijski mehanizmi so znani kot "vstopi" (entries) in "pogoji" (conditions).

Primer:

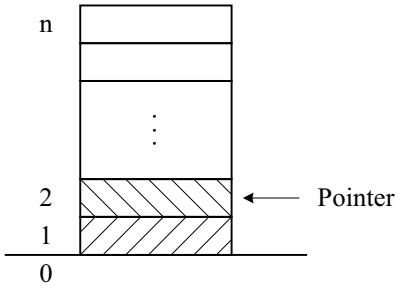
```
   P1                      P2
  -------                 -------
Buffer:WriteData(x)     Buffer:ReadData(x)
  -------                 -------
```

Oba klica sta enolično se izključujoča, zato se morata sinhronizirati.

Uporaba monitorjev je enostavnejša kot uporaba semaforjev. Procesi, ki se morajo med seboj uskladiti - sinhronizirati, enostavno kličejo vstop v monitor, seveda opremljeni z ustreznimi parametri, ki pa v bistvu predstavljajo le klice procedur. Najpomembnejše je to, da se ne more zgoditi, da bi v programu pozabili izvesti postopek P ali V. Seveda nastavitev parametrov pri vstopu v monitor ni tako enostavna, kot bo razvidno iz primera. Pri tem seveda operiramo s skladovnico "stack buffer".

Pri vstopu v sam monitor najprej določimo monitorjeve podatke. Ti imajo sledeče lastnosti:

- statičnost (monitor ohranja podatke ob vstopu med klici monitorja),



Slika 5.12: Organizacija skladovnika

- lokalnost (monitorjevi podatki so dosegljivi le s klici monitorjevih vstopov).

Deklaracija vstopov, monitorjevih procedur, je naslednja. To so deli monitorja, ki se izpostavijo in se jih lahko kliče s procesi kot parametre (VAR). Ker dostopajo k porazdeljenim podatkom, so ekskluzivno enolični v celotnem času izvajanja. Za zahtevano sinhronizacijo poskrbi operacijski sistem, kar pomeni, da vse dokler se proces nahaja znotraj monitorskega klica, tedaj morajo vsi procesi, ki žele klicati vstop v isti monitor, čakati. Sekcija inicializacije se postavi na koncu monitorja in se izvede le enkrat in sicer v času zagona celotnega programskega sistema, včasih celo pred glavnim inicializacijskim programom.

Pogoji so tu vrste, kot so vrste pri semaforjih. Seveda vrsta tu ni predstavljena s števcem, kot je to pri semaforjih. Na pogojih se izvedejo sledeče tri operacije:

wait (cond) Klicajoči proces blokira in čaka, vse dokler drugi proces ne izvede operacije "signal" (cond) nad istimi pogoji

signal (cond) Vsi procesi, ki čakajo pri tem neizpolnjenem pogoju "cond", se oživijo in ponovno zahtevajo dostop do monitorja. (Druga možnost je tudi, da se reaktivira - oživi le en proces, drugi počakajo v vrsti).

status (cond) Ta operacija da število procesov, čakajočih na izpolnitev njihovih posebnih pogojev (cond). (Število blokiranih procesov).

Oglejmo si primer programa za upravljanje "buferja" sklada v Moduli P. V tem jeziku sta uporabljena dva pogoja "free" in "occupied". Procesi v vrsti čakajo na izpolnitev njihovih posebnih pogojev.

V primeru, da se želi izvesti vstop "WriteData" in da je izraz "Pointer = $n \equiv$ true", kar pomeni, da je sklad poln, tedaj morajo klicajoči procesi čakati, da se vsaj eden od "buferjev" sprosti. To čakanje se izvede znotraj zanke "while". Temu sledijo operacije nad podatki monitorja (vpis parametra a). Nazadnje signal na pogoju "occupied" aktivira poljuben proces, ki morda čaka na vstopu "ReadData".

Slično konstrukcijo ima vstop "ReadData". Če je sklad prazen, tedaj mora proces počakati vse dotlej dokler se ne pojavi pogoj "occupied", nakar šele lahko bere podatke s sklada. Na koncu se izvede signal na pogoju "free", ki aktivira vse procese, čakajoče v "WriteData" vstopu. Ker so signali zahtevne operacije, se jih vključi v "if" testiranje in se jih uporabi kadar obstajajo čakajoči procesi.

```
monitor Buffer;
var     Stack: array[1..n] of Datarecord;
        Pointer: 0..n;
        free, occupied: condition;

entry WriteData (a:Datarecord);
begin
  while Pointer=n do (*stack full*)
    wait(free)
```

```
    end;
    inc(Pointer);
    Stack[Pointer]:=a;
    if Pointer=1 then signal(occupied) end;
  end WriteData;

  entry ReadData (var a:Datarecord);
  begin
    while Pointer=0 do (*Stack empty*)
      wait(occupied)
    end;
    a:=Stack[Pointer];
    dec(Pointer);
    if Pointer=n-1 then signal(free) end;
  end ReadData;

  begin              (*Monitor Initialization*)
    Pointer:=0;
  end monitor Buffer;
```

Monitorji se v splošnem uporabljajo skupaj s semaforji. Uporaba je v splošnem izvedena v štirih korakih:

1. Boolova spremenljivka semaphore je deklarirana za vsak deklariran monitor.

   ```
   var MSema: semaphore[1]
   ```

2. Vstopi se transformirajo v procedure, kjer je vsaka objeta z operacijama P in V na monitorju semaforja. To pa pomeni, da lahko le en sam proces izvaja ukaze na vstopu.

   ```
   procedure xyz(...);
   begin
     P(MSema):
       <Instructions>
     V(MSema);
   end;
   ```

3. Inicializacija monitorja se prevede v proceduro. Vsak klic le-te se določi na začetku glavnega programa.

4. Uporaba operacije “wait”

   Klicajoči proces se doda vrsti in se blokira. Nakar se sprosti monitor semaforja in naslednji proces v stanju Ready se lahko izvede. Vzporedna konstanta actproc pa kaže na število trenutno aktivnih procesov.

5.3 Izmenjava podatkov v strukturah SIMD

Izmenjava podatkov na SIMD računalniku se izvede vzporedno na vseh aktivnih PE. Takšna izmenjava poteka v paru med PE in je lahko lokalna ali globalna. Izvajanje se izvrši v treh korakih:

1. Izbira skupine PE (aktivacijski postopek).

2. Izbira predhodno določene smeri povezav v sistemu povezav ali določitev nove dinamične strukture sosedov.

3. Izvajanje prenosa podatkov med izbranimi strukturami povezav.

Tovrstna izmenjava podatkov je predstavljena v zato razvitih jezikih primernih za strukture SIMD. Takšen jezik je PARALLAXIS (BRÄUNL - University Kaiserlauterm Germany). Nekateri jeziki pa vsebujejo direktne funkcije povezav kot direktne aritmetične operacije.

Primer v jeziku Parallaxis:

1. Določitev strukture povezav:

```
Ex.:  Configuration ring[0..11];
      Connenction  right: ring[i]<->[ring[i+1] modR].left;
```

   Struktura povezav ring vsebuje 12 elementov $[0, \ldots, n-1]$. V sistemu obstajajo dvosmerne povezave s simboličnima imenoma left, right. Povezava na desno preslika vsak PE na proc. element z višjim indeksom. PE11 se preslika v PE0 po modulu. Funkcija povezav daje ring.

2. Izbira skupine PE

```
Ex.:  Parallel ring[3..8]
      ----------
      EndParallel
```

   V tej skupini procesorjev, ki smo jih izbrali so PE aktivni, ostali so pasivni.

3. Izvedba prenosa podatkov (paralelno) (znotraj paralelnega bloka).

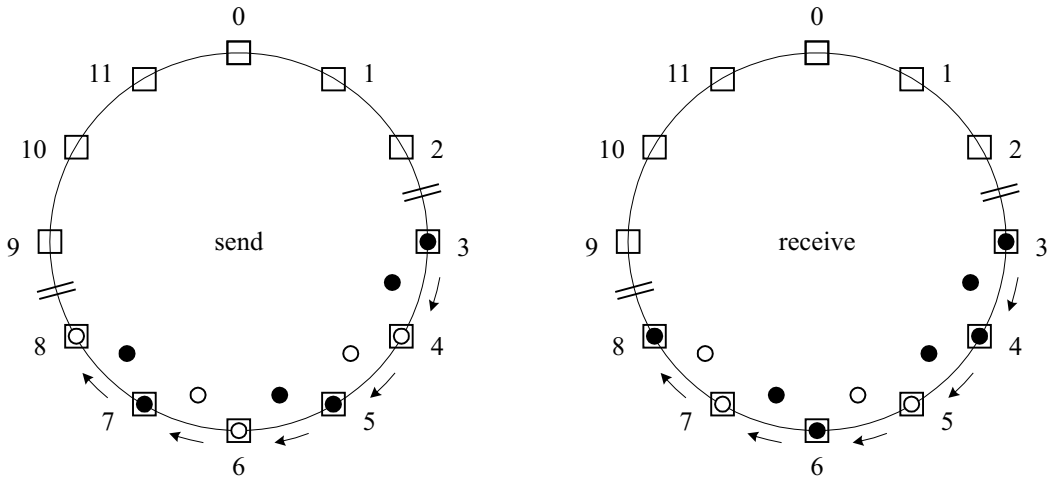
```
Ex.:
     Propagate.right(x)
```

Vsak od aktivnih procesorjev izvede prenos podatkov k svojemu sosedu. Ker je to struktura obroča se ta izvede v smeri urinega kazalca. Vsak PE najprej postavi njegovo lokalno spremelnjivko na povezovalno vodilo (send), nakar prebere novo vrednost lokalne spremenljivke njegovega predhodnika. Ker smo aktivirali le del elementov na obroču, element PE 3 nima predhodnika in element PE 8 nima naslednika. Isto se zgodi v odprtih linearnih strukturah. Zato v tem primeru PE 3 ohrani staro vrednost, dočim PE 8 prejme lokalno spremenljivko od PE 7.

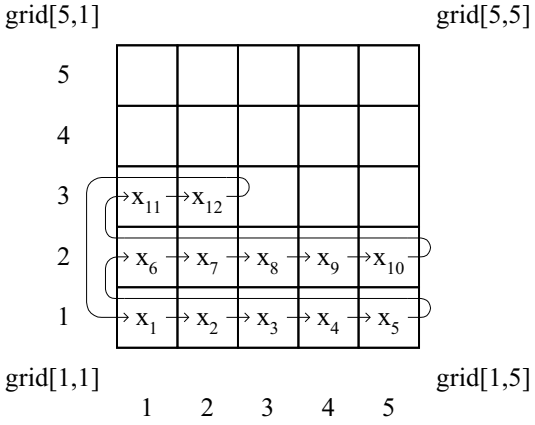
Tovrstno strukturo obroča lahko enostavno preslikamo v vse bolj poznano mrežno povezovalno strukturo. Kot so tudi v strukturi obroča sosedje med seboj povezani, tako je tudi s povezavami v mreži. V tem primeru je lahko skrajni element na desni povezan s skrajnim elementom na levi, ki pa je lahko ali vrstico nižje ali v isti vrstici. Tudi v mrežni strukturi lahko pride do povezane omenjene strukture obroča.

```
Ex.:   Virtual Structure:
         Parallel ring [0..11]
           Propagate.right(x)
         Endparallel
```

To preslikamo v fizično mrežo:



Slika 5.13: Struktura obroča



Slika 5.14: Virtualni obroč

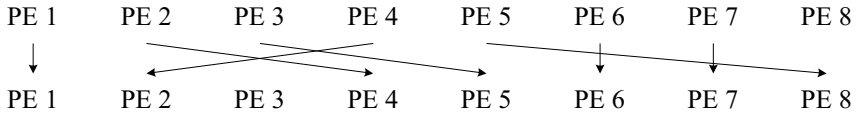
```
Parallel grid[1..2],[1..4]; primer a: korak na desno
         grid[3]  [1]
'grid[i,j] -> grid[i,j+1]'
End Parallel

Parallel grid[1..2],[5];   primer b: povezi na zacetek
'grid[i,j] -> grid[i+1,1]'           naslednje vrstice
EndParallel;

Parallel grid[3],[2];      primer c: povezava na zacetek
'grid[i,j] -> grid[1,1]'
EndParallel;
```

V mejnih primerih se morajo podatki zamenjati virtualno. Zato se morajo prireditve izvršiti izven področja "buferja", drugače se lahko podatki prepišejo s podatki svojega soseda.

Kot smo videli, obstaja avtomatska preslikava virtualnih povezav na fizično strukturo, vendar izvedba le-te ni enostavna. Avtomatsko povezavo procesorjev lahko izvedemo le za enostavne strukture. Za kompleksnejše strukture pa si bomo ogledali postopek preslikave kasneje.

Čim narašča kompleksnost virtualnih struktur (algoritma), je postopek preslikave težji in težji. Avtomatske preslikave tudi niso smiselne za kompleksnejše strukture, ampak so lahko le v pomoč.

V literaturi zasledimo tudi globalno univerzalne povezovalne strukture ("router") za SIMD arhitekturo. Naslov ponora (povezave) procesorskega elementa PE je določen za vsak fizični PE. Tako si lahko razlagamo posplošen primer nestrukturiranih povezav kot permutacijo vektorjev.



Slika 5.15: Virtualna povezava procesorskih elementov

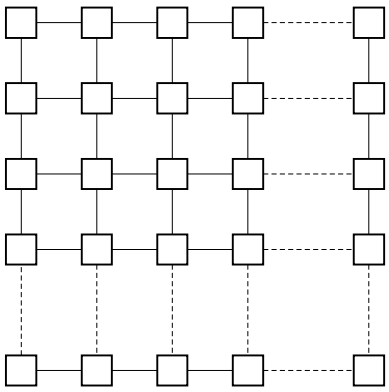
Uporaba globalnih povezovalnih struktur je enostavna in jo lahko brez težav tudi avtomatiziramo. Globalna povezovalna struktura je v principu počasnejša od lokalnih mrežnih povezav. Pravilna uporaba globalnih ali lokalnih povezav, pa predstavlja enega resnejših problemov v strukturah SIMD. Kompleksne in nestrukturirane topologije zahtevajo vpeljavo virtualnih procesorjev med realne procesorje PE. Zato se v teh primerih potrebujejo posebni vmesni pomnilniki, ki nadomeste vse virtualne procesorje, ki pa zato še povečujejo kompleksnost preslikave virtualnih struktur na realne PE.

5.3.1 Primeri povezav v SIMD sistemih

Kot smo že omenili Connection Mashine CM-2 in MasPar MP-2 spadata v razred SIMD arhitektur. Oba sistema poznata globalni sistem povezav, ki omogoča povezavo s poljubnim izbranim parom procesorjev, glede na lokalno povezovalno mrežo, ki pa omogoča le povezavo z najbljižjim procesorjem. Lokalne mrežne povezovalne strukture so očitno hitrejše in so uporabne za reševanje problemov slik, globalne strukture omogočajo izbiro dinamično izbranih povezovalnih topologij.

Connection Machine CM-2

1. Mrežna struktura CM-2 (65.536 PE) je dinamična mreža s štirikratnimi povezavami s svojimi sosedi ("NEWS"- North, East, West, South), ki jo lahko prevezujemo znotraj fizičnih omejitev v sisteme poljubnih struktur. Hitra povezovalna struktura je uporabljena kot del globalne strukture hiperkocke, s posebno strojno izvedbo. Dvodimenzionalna struktura povezav (256×256 PE) je prikazana na sliki 5.16.

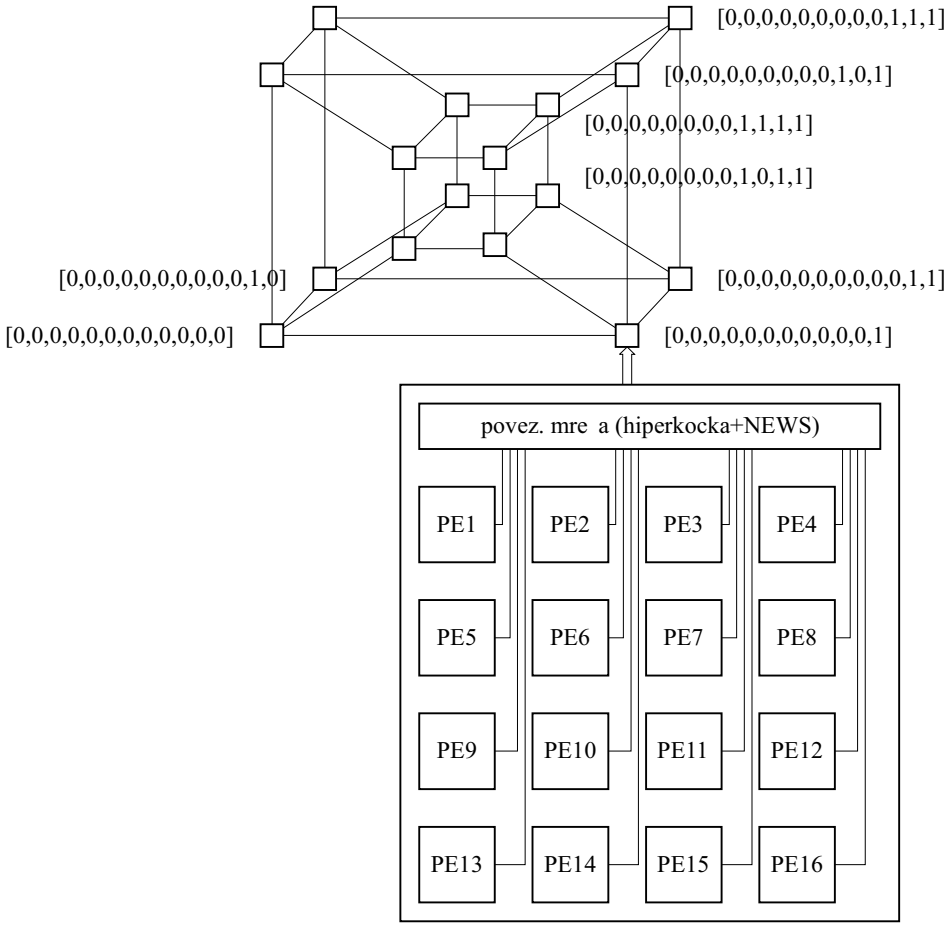


Slika 5.16: Slika povezav PE v CM-2

2. Dvanajstdimenzionalna hiperkocka je uporabljena kot globalna povezovalna struktura v CM-2.

Glede na to, da ta hiperkocka vsebuje le $2^{12} = 4096$ PE, moramo torej 65.536 elementov razdeliti na 4096 grozdov, kjer vsak vsebuje 16 PE, torej na štiridimenzionalno hiperkocko s slike 5.17.

Če mora kakšen od PE komunicirati izven grozda, tedaj se ta celoten postopek izvede v 16 korakih, zaradi omejitev na komunikacijskem vodilu.



Slika 5.17: Štiridimenzionalna hiperkocka

Do neke meje je globalna povezovalna struktura v CM-2 omejena. Vsekakor pa dopušča konstrukcijo vzporednega računalnika s 64 procesorji, pri čemer je število povezav še znosno. Rešitev v obliki grozda torej zahteva

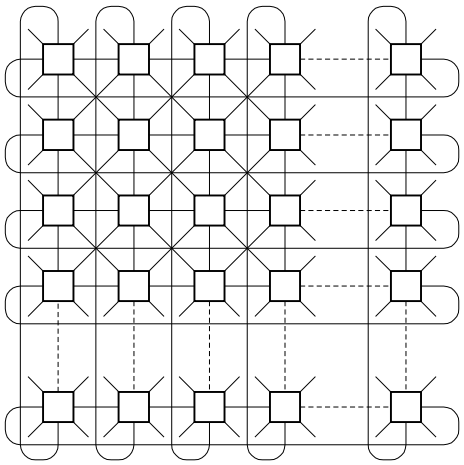
$$\begin{aligned} \frac{1}{2} \cdot 4096 \cdot 12 &= 24576 \text{ povezav v hiperkocko} \\ 65536 \cdot 1 &= 65536 \text{ povezav posameznih elementov v grozdu} \\ &\quad - - - - - - - \\ &\quad 90112 \text{ povezav} \end{aligned}$$

V primeru, da pa bi želeli realizirati popolno 16 dimenzionalno hiperkocko, pa potrebujemo

$$\frac{1}{2} \cdot 65536 \cdot 16 = 524288 \text{ povezav}$$

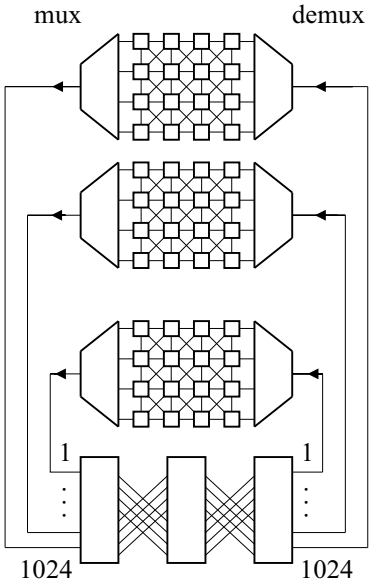
MasPar - Model MP 2216 s 16384 PE

MP-2 struktura je fiksna lokalna mreža z 8 povezavami na PE in je urejena kot mreža 128×128 PE povezanih v dvojni cilinder. Povezave posameznih PE so v smereh NEWS in smereh diagonal X.



Slika 5.18: Struktura zaključenih povezav

MP-2 ima tudi globalno povezovalno strukturo, ki je sestavljena iz trostanjskega stikalnega omrežja s 1024 vhodi in izhodi. To pomeni, da si grozd 16 PE deli en povezovalni vhod in en izhod. Od 16384 PE ima le 1024 PE direktne povezave s povezovalnikom. To vezje je cenejše od klasičnega CROSSBAR stikalnika, ki je za 16384 PE za 144-krat dražji glede na predlagano rešitev v MP-2.

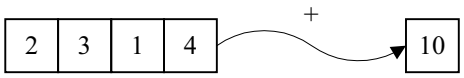


Slika 5.19: MP-2

Redukcija vektorja

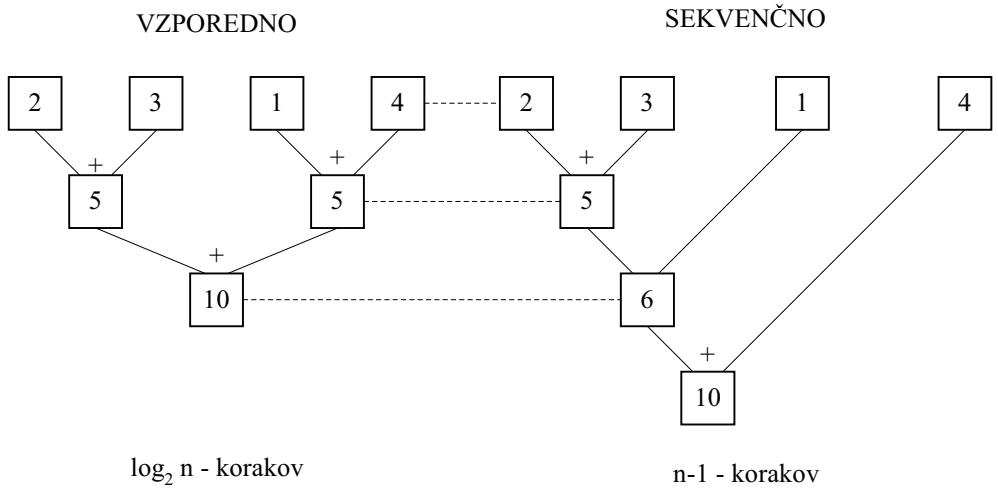
Osnovna operacija v vektorskih računalnikih ali SIMD sistemih, je redukcija vektorja. Včasih naletimo v določenih sistemih to operacijo kot primarno operacijo, v nekaterih sistemih pa celo naletimo na posebno strojno rešitev. Tu vektor reduciramo na skalarno veličino. To izvedemo z dvooperandnimi ukazi kot so seštevanje, množenje, maksimum, minimum, logična AND in OR itd. Redukcija je asociativna in komutativna operacija, saj bi drugače dobili za razne vrstne rede izvajanj, različne rezultate.

Komponente vektorja seštejemo. Rezultat je skalarna veličina. Vrstni red izvajanja ni važen, zato



Slika 5.20: Redukcija vektorja

npr. tronivojsko vzporedno izvajanje omogoča učinkovitejše rešitve kot pa sekvenčno izvajanje.



Slika 5.21: Primera redukcije vektorja

V nasprotju s sekvenčnim seštevanjem, ki zahteva $n-1$ korakov, vzporedna drevesna struktura omogoča pri istem številu operacij seštevanje le v $\log_2 n$ korakih (nivojih).

V notaciji jezika Parallaxis je redukcija vektorja $\vec{v}$ izvedena kot:

```
s: = Reduce.Sum(V)
```

Vektorska redukcija se lahko izvede programsko ali z uporabo posebne strojne opreme, kar omogoča redukcijo števila vzporednih operacij pri prenosu podatkov ali izračunu posameznih algoritmov.

5.3.2 Problemi

Kot smo videli je asinhrono vzporedno programiranje relativno zahtevno in je torej nagnjeno k napakam. Pozabljanje operacij P ali V ali celo uporaba napačnega semaforja, je tipičen in čest primer napak, ki povzroče težave v izvajanju programa. Pri monitorjih obstaja nevarnost nepravilne uporabe pogojnih spremenljivk, ali celo operacij "wait" in "signal". Napake se torej pojavljajo predvsem na nivoju sinhronizacije procesov, kar ima za posledico nepravilne podatke oziroma zaklepanje sistema.

Prav tako obstaja problem ne pravilnega razvrščanja nalog po procesorjih (load balancing), saj bi vedno radi optimalno izkoristili procesorsko moč.

Največ problemov pri sinhronem paralelizmu pa nastopi predvsem zaradi omejitev modela SIMD.

Ker večina procesorjev opravlja isto operacijo ali so celo neaktivni, se lahko zgodi, da določene operacije niso dovolj paralelizirane. Uvedba določenega nivoja abstrakcije in s tem virtualnih procesorjev neodvisno od obstoječega števila dejanskih PE lahko tudi povzroča probleme. Naslednji zelo pomemben problem pri SIMD sistemih pa je povezava s perifernimi enotami, kar povzroča ozko grlo pri prenosu podatkov.

V tem poglavju si oglejmo predvsem probleme s sinhrono vzporednostjo.

Indeksirane vektorske operacije

Izraza "gather" (zbiranje) in "scatter" (raztros) označujeta dve osnovni vektorski operaciji, ki povzročata probleme pri uporabi sinhronega paralelizma. Problem dejansko nastopi pri vektorizaciji dostopa indeksiranih podatkov.

```
Gather:                    Scatter:
for i:=1 to n do           for i:=1 to n do
  a[i]:=b[index[i]]          a[index[i]]:=b[i]
end;                       end;
```

Vsi vektorji se porazdele po PE tako, da vsakemu PE pripada en element vektorja. Operacija "gather" bere vektorje, ki so indeksirani z drugim vektorjem, dočim operacija "scatter" vpisuje vektor, kateri je indeksiran z drugim vektorjem.

V obeh primerih so podatki odvisni od vrednosti indeksnega vektorja, kar lahko povzroči nekontrolirane zamenjave podatkov in naključnost dostopa do podatkov. Takega obnašanja pa se ne da paralelizirati s strukturo procesorjev povezanih v mrežo, kjer algoritmi delujejo z izmenjavo podatkov z najbližjimi procesorji.

V vektorskih računalnikih se za tovrstne naključne primere uporabljajo posebna vezja in ne časovno zahtevni programski prenosi. Indeksno zasnovan dostop do podatkov torej običajno uporablja počasne povezovalne strukture v sistemih masivnega vzporednega računanja. Drugače pa nam le še preostane prenos indeksiranih podatkov preko sosedov, kar pa je časovno zelo potratna operacija. Celo indirekten dostop do vektorja lahko povzroči na istem PE SIMD sistemu velike težave.

```
Ex.:   SCALAR s: INTEGER;
       VECTOR a: ARRAY[1..10] OF INTEGER;
            u,v: INTEGER
       --------
       u:= a[s]; (*scalar index*)
       u:= a[v]; (*vector index*)
```

Dejansko gre za prireditev niza vektorja a vektorju n. Uporabe vektorskega indeksa za naslavljanje vektorskega niza (vsak PE ima drug indeks) ni mogoče realizirati v Connection Machine CM_2 zaradi pomanjkanja ustrezne strojne opreme, je pa ta način možen v sistemu MasPar MP-2.

Problem preslikave virtualnih procesorjev na realne procesorje

Vpeljavo tega nivoja abstrakcije si še oglejmo, predno preidemo na težave s preslikavo virtualnih procesorjev.

V SIMD računalnikih imamo na razpolago veliko množico PE. Kljub vsemu temu, da imamo npr. 65536 procesorjev, pa je to število premajhno za obdelavo navadne slike z npr. $500 \times 500 = 250.000$ točkami. Ker preslikavo le-teh na fizične procesorje izvede programer sam, moramo vpeljati nekakšne virtualne procesorje, ki jih nato glede na program preslikamo na realne procesorje v nekakšnih iteracijah. Te iteracije izvedemo lahko na nivoju grupe ukazov ali ukaza samega. Če je ν število virtualnih procesorskih elementov VPE in je p število realnih procesorskih elementov PE, je stopnja virtualizacije $R = \frac{\nu}{p}$. Če seveda ta faktor ni celo število, pride do določene stopnje izgube paralelnosti.

Izmenjava podatkov je kompleksna posebno pri zahtevnejših strukturah med VPE in PE. Z vsako izmenjavo podatkov se preračunavanje seveda upočasni.

Preslikava mora biti jasna vsakemu programerju. To tudi pomeni, da je lahko izvedena s programsko ali strojno opremo.

1. Preslikava v hardware-u

CM-2 Connection Machine ima ustrezno strojno rešitev za delo z VPE. Stopnja virtualizacije se lahko direktno določi. Glavni pomnilnik vsake PE se razdeli med vse VPE in za vsak VPE se izvaja interativno isti ukazni niz. Ta rešitev ima določene prednosti pred programsko, saj omogoča učinkovito vzporedno procesiranje, programi pa so enostavnješi. Seveda so pa ti "elementi" neuporabljeni, če ni zahtevane $R = 1$.

2. Preslikava s programsko opremo

   Prevajalnik mora zagotoviti virtualne procesorje z izvedbo naslednjih nalog:

   - določiti število VPE in PE in določi faktor R,
   - določiti lokacije za podatke VPE

     ```
     (VECTOR i: INTEGER; => VECTOR i: ARRAY[1..R] OF INTEGER;
     ```

   - generiratie zanke za določitev vektorjev

     ```
     i:=2*i;          =>  FOR STEP:=1 TO R DO
                            i[STEP]:=2*i[STEP];
                          END;
     ```

   - zaradi izmenjave podatkov prevesti VPE naslove realnih PE.

MasPar MP-2 ima virtualizacijo rešeno programsko z uporabo "intelegentnega prevajalnika". Tovrsten prevajalnik generira posebno kodo, ki določa ali je virtualizacija potrebna ali ne. Tako uporabnik ne zazna razlike med obema rešitvama, pa čeprav je programska rešitev mnogo zahtevnejša.

Tak pristop virtualizacije omogoča neomejeno število VPE v poljubnih povezavah. Samo z abstrakcijo strojne opreme pridemo do strojno neodvisnih vzporednih programov. Pri preslikavi pa se srečamo s sledečimi problemi:

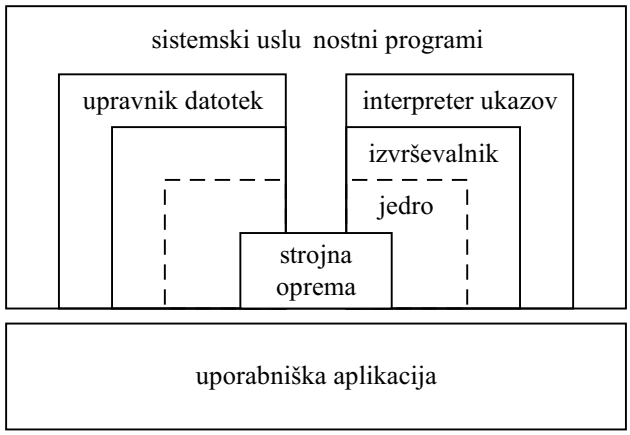
- virtualni PE se dele med realne PE,
- če je le možno se uporabljajo najhitrejše povezave.
- avtomatsko generiranje optimalne preslikave virtualne topologije na realne PE.
- če potrebujemo več VPE kot je rednih PE, tedaj se izmenjava podatkov lahko izvrši le preko velikega vmesnika (pomnilnika) z določeno rezervo. Vse to pa upočasni delo.

Izmenjava podatkov v VPE je torej najkritičnejši element in je rešljiva za kompleksne podatkovne strukture le v primeru, če obstaja ustrezna struktura povezav.

Poglavje 6

Operacijski sistemi za delo v realnem času

Funkcije OS za delo v realnem času so porazdeljene v več nivojev (slika 6.1).



Slika 6.1: Funkcije OS za delo v realnem času

Centralni del se imenuje *jedro* in vsebuje procedure, ki podpirajo sočasno izvajanje opravil. Jedro je rezidenčni program, ki deluje pod nadzorom *izvrševalnika*. Vse sistemske aktivnosti, ki jih podpira jedro, vodi izvrševalnik. Zaporedje izvajanja opravil ne določa program, temveč izvrševalnik glede na dogodke, saj so namenski sistemi za delo v realnem času dogodkovno in ne ukazno vodeni.

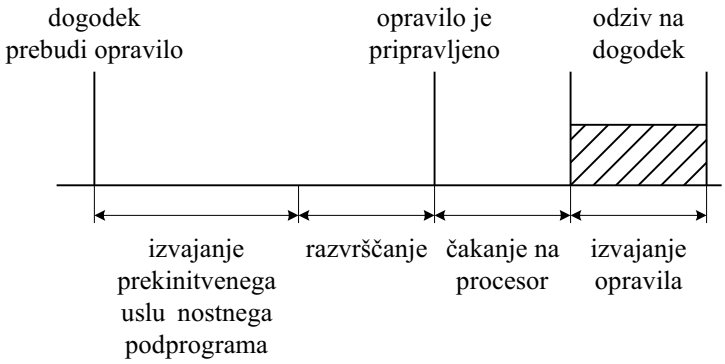
Procedure jedra morajo ustrezati zahtevam realnega časa, kot so pravilnost, časovna omejenost, zanesljivost in varnost. Osnovne funkcije teh sistemskih procedur so sledeče:

1. razvrščanje opravil (*task scheduling*),
2. upravljanje skupnih virov (*mutual exclusion*),
3. komunikacija med opravili,
4. naslavljanje pomnilnika ter
5. izogibanje napakam (*fault-handling*).

Opravila predstavljajo enote programskih aplikacij, ki so pogojene s programsko (*re-entrant*) kodo, lokalnimi skupnimi podatki, skladom, oznako opravila ter nadzornim blokom (*task control block*). Če dogodek sproži večkrat opravilo, potem si "kopije" opravila delijo oznako in programsko kodo. Nadzorni blok hrani vse podatke o opravilu.

6.1 Razvrščanje opravil

Modul OS za delo v realnem času, ki izvaja razvrščanje opravil, imenujemo razvrščevalnik. Njegova naloga je določiti vrstni red izvajanja *prebujenih opravil*. Opravila so prebujena, če stanje sistema zahteva njihovo izvajanje (slika 6.2).



Slika 6.2: Dogodek sproži izvajanje opravil

Ker je običajno število prostih virov računalniškega sistema manjše od števila prebujenih opravil, je pomemben vrstni red izvajanja opravil. Prednost imajo opravila, ki so časovno najbolj omejena in pomembna (kritična) za aplikacijo. Stopnja pomembnosti in večina časovnih lastnosti opravil so določene pred izvajanjem aplikacije.

Vidiki razvrščanja

Pri razvrščanju opravil računalniških sistemov za delo v realnem času mora razvrščevalnik uporabiti več vidikov:

1. Izvajanje opravil je časovno omejeno.
2. Opravila so različno pomembna za aplikacijo.
3. Viri sistema morajo biti dobro izkoriščeni.

Našteti vidiki se medsebojno izključujejo. Optimizacija enega vidika lahko poruši zadostitev ostalih dveh. Pri namenskih sistemih za delo v realnem času je najpomembnejši prvi vidik. Vsa opravila aplikacije so časovno omejena in se morajo izvršiti pravočasno.

Za razliko od sistemov na liniji, ki skušajo doseči najboljši možni povprečni čas odziva na dogodke, namenski sistemi dajejo prednost pravočasnemu izvajanju kritičnih opravil.

Vidik izkoriščenosti virov sistema moramo v marsikateri namenski aplikaciji zanemariti. Ker lahko asinhron dogodek sproži zahtevo za izvajanje kritičnega opravila (npr. alarm), je pomembno, da so zahtevani viri prosti v tistem trenutku.

Cena opreme računalniškega sistema sicer pri tem močno naraste, čemur se zlasti pri varnih aplikacijah ne moremo izogniti. *Izkoriščenost vira* (U) lahko definiramo kot čas, ki ga vir porabi za izvajanje (aplikacijskih in sistemskih) opravil:

$$U = \frac{\sum_{i=1}^{m} T_{Ci}}{T},$$

pri čemer T_{Ci} pomeni čas izvajanja opravila τ_i iz nabora m prebujenih opravil ter T čas izvajanja aplikacije.

Optimalno in idealno razvrščanje

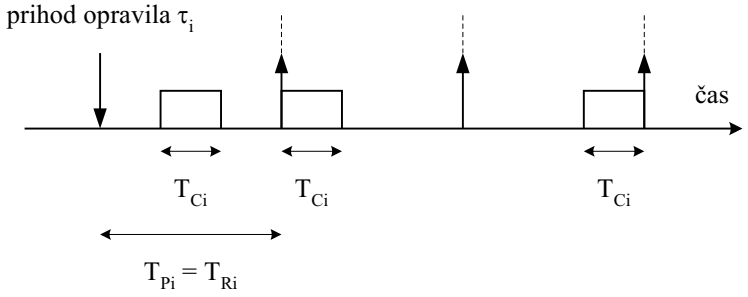
Razvrščevalnik določi zaporedje izvajanja opravil po izbranem *principu razvrščanja*. Razvrstitev opravil je lahko *optimalna* ali *idealna*. V prvem primeru je zagotovljena minimalna zakasnitev opravil, kar pomeni, da je možna prekoračitev časovne omejenosti nekaterih, običajno manj pomembnih opravil. Idealna razvrstitev pa zagotavlja odziv sistema brez zakasnitev.

Opravila, ki zadoščajo pogojem optimalne in idealna razvrstitve čakajo na dostop do procesorja v vrsti pripravljenih opravil.

Pogoj kateremu mora razvrstitev m pripravljenih opravil za dani vir ustrezati, da jo označimo kot idealno, je:

$$\sum_{i=1}^{m} \frac{T_{Ci}}{T_{Pi}} \leq 1.$$

Vendar moramo poudariti, da velja pogoj pri določenih predpostavkah. Nabor opravila zajema le periodična opravila. T_{Pi} pomeni periodo opravila τ_i, katerega čas izvajanja je enak T_{Ci}. Dodaten pogoj, ki sta ga avtorja uvedla, je izenačitev ozivnega (*latency*) časa (T_{Ri}) opravila τ_i ($i = 1, \ldots, m$) z njegovo periodo (T_{Pi}) (slika 6.3).



Slika 6.3: Periodično opravilo

Pogoj smo razširili tudi za nabor m aperiodičnih opravil:

$$U = \frac{\sum_{i=1}^{m} T_{Ci}}{T_{Dm}} \leq 1. \tag{6.1}$$

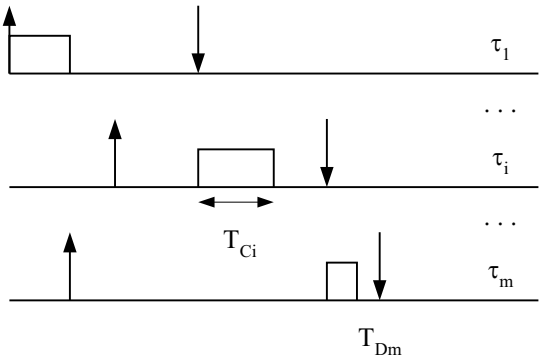
kjer je T_{Ci} čas izvajanja opravila τ_i in T_{Dm} čas, ko mora opravilo τ_m zaključiti z izvajanjem, sicer bo zavrnjeno. Opravila so urejena glede na časovno omejenost izvajanja (slika 6.4). Opravilo τ_1 je časovno najbolj kritično. Zato se izvaja brez zakasnitve, medtem ko se opravilo τ_m, kot najmanj kritično, izvaja z največjo zakasnitvijo.

Omejitve pri izboru razvrščevalnega principa

Razvrščevalni princip je skupek pravil, s pomočjo katerih razvrščevalnik določa razvrstitev prebujenih opravil. Izbor principa temelji na zahtevah aplikacije. Nekatera jedra nudijo več procedur, ki uporabljajo različne razvrščevalne principe. Ena izmed nalog izvrševalnika je tudi dinamično preklapljanje med principi.

Faktorji, ki so pomembni pri izboru principa, so sledeči:

- vrsta opravil,
- relacije med opravili,
- zahtevana stopnja zanesljivosti,



Slika 6.4: Urejenost aperiodičnih opravil glede na njihovo časovno kritičnost

- časovna omejenost posameznih opravil,
- kritičnost opravil in
- omejitev virov.

Opravila namenskih aplikacij so lahko periodična ali aperiodična. Pogosto namenske aplikacije, ki se izvajajo v realnem času, združujejo obe vrsti opravil.

Opravila so lahko medsebojno odvisna ali neodvisna. Odvisnost je podana z dvema relacijama, relacijo izključevanja (*exclude relation*) in relacijo zaporedja (*precedence relation*). Relacija izključevanja določa za vsako odvisno opravilo množico opravil, ki ne smejo prekiniti njegovo izvajanje. Relacija zaporedja omejuje možnost razvrščanja opravil, saj določa, katero opravilo mora biti predhodno izvršeno, da se lahko pričnejo izvajati določena ostala opravila.

Kot smo že omenili, mora biti stopnja zanesljivosti izračuna opravil izredno visoka, zlasti v primeru varnih sistemov. Zadostitev tega pogoja ni odvisna le od izbora učinkovitega razvrščevalnega principa, temveč tudi od vrste drugih faktorjev. Zahtevani razvrščevalni principi so lahko časovno in prostorsko potratni. Podpora koprocesorja, ki prevzame nadzor nad izvajanjem sistemskih opravil, rešuje omenjeni problem. Vendar pa zahteva takšna rešitev drugačen pristop k oblikovanju celotnega namenskega sistema.

Opravila se običajno razlikujejo glede na zahtevano hitrost odziva in pomembnost (kritičnost) za aplikacijo. Pravimo, da imajo opravila različne časovne in logične prioritete. V namenskih aplikacijah sta obe prioriteti pomembni, kar mora upoštevati razvrščevalni princip. Zlasti logične prioritete opravil so subjektivne narave, saj jih določa programer po lastni presoji. Časovne prioritete izražajo časovne lastnosti opravil (slika 6.5):

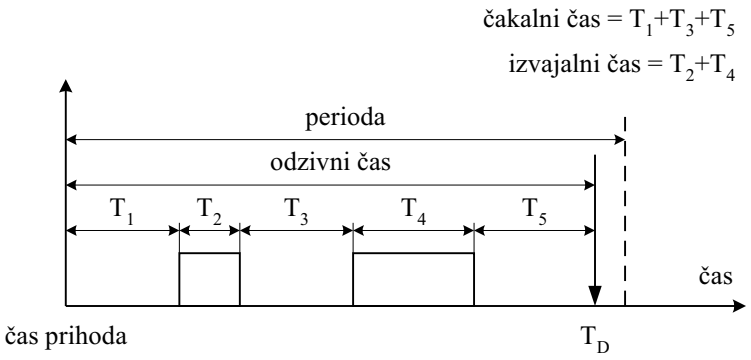
- čas prihoda (T_A),
- predvideni čas izvajanja (T_C),
- zahtevani odzivni čas (T_R),
- perioda (T_P).

Čas prihoda je določen s trenutkom, ko se je spremenilo stanje sistema in je dogodek sprožil zahtevo za izvajanje opravila. *Čas izvajanja* opravila je pogosto odvisen od vhodnih podatkov, zato mora programer predvideti najdaljši možni čas izvajanja opravila. Sistem za delo v realnem času zahteva odziv na dogodek v predvidenem intervalu, ki ga imenujemo *odzivni čas*. Odzivni čas določa *časovno omejitev* T_D:

$$T_D = T_A + T_R \tag{6.2}$$

Pomemben je tudi podatek o *preostalem čakalnem času* (T_L) v času t. Ko preteče čakalni čas, se opravilo gotovo ne bo izvršilo pravočasno.

$$T_L = T_D - t - T_C(t) \tag{6.3}$$



Slika 6.5: Časovne lastnosti opravil

$T_C(t)$ podaja preostali čas izvajanja opravila v času t. Perioda določa časovni interval, v katerem se ponovi zahteva za izvajanje periodičnega opravila.

Principi razvrščanja se prilagajajo vrstam virov (npr. procesorjem, vhodno/izhodnim perifernim enotam). Poleg zmogljivosti in prioritet posameznih virov, mora princip upoštevati tudi omejitev v številu razpoložljivih virov za izvajanje opravil.

6.1.1 Principi razvrščanja pri enoprocesorskih sistemih

Tabela 6.1 podaja pregled principov razvrščanja, ki se uveljavljajo pri operacijskih sistemih za delo v realnem času. Principe smo razdelili glede na način razvrščanja, ki je lahko:

- enostavno ali priotetno,
- brez ali s prekinjanjem,
- statično ali dinamično.

Enostavno razvrščanje ne upošteva prioritet opravil, temveč jih obravnava povsem enakovredno. Principi s prioritetnim razvrščanjem so bolj zapleteni, vendar nudijo večjo prilagodljivost sistema na dogodke. Nekateri principi obravnavajo časovne ali logične prioritete opravil. Najbolj učinkoviti so tisti principi, ki razvrščajo opravila glede na časovne in logične prioritete.

Principe nadalje delimo na tiste, ki podpirajo razvrščanje brez prekinjanja in s prekinjanjem. Prva skupina principov ne dovoljuje prekinjanja izvajanja opravil, katerim je bil dodeljen procesor. Takšni principi niso primerni za namenske sisteme za delo v realnem času, katerih stanje se dinamično spreminja. V primeru pojavitve dogodka, ki zahteva nenaden odziv sistema, mora prebujeno opravilo čakati, da se procesor sprosti, ne glede na kritičnost opravila, ki se trenutno izvaja. Principi s prekinjanjem so bolj učinkoviti, a tudi časovno bolj zamudni, saj zahtevajo dodaten čas za shranjevanje trenutnega stanja sistema (*context switching*). Prekinjeno opravilo nadaljuje z izvajanjem, ko se procesor ponovno sprosti.

Principe s prioritetnim razvrščanjem, ki podpirajo prekinjanje izvajanja opravil, delimo na statične in dinamične. **Statični** principi razvrščajo opravila glede na njihove lastnosti, ki se med izvajanjem

Principi	brez prekinjanja	s prekinjanjem
enostavni	FCFS	krožno dodeljevanje
prioritetni	prednost krajših oprtavil	statični
	prednost čakajočih opravil	dinamični

	statični	dinamični
subjektivni	prednost kritičnih opravil	prednost časovno kritičnih opravil
objektivni	prednost krajših opravil	
	prednost opravil s krajšim odzivnim časom	
	prednost pogostejših opravil	

Tabela 6.1: Preglednica razvrščevalnih principov

aplikacije ne spreminjajo. Lastnosti opravil so lahko subjektivne narave, kot je npr. kritičnost, ki jo določi programer po lastni presoji. Časi izvajanja in odzivni časi so objektivne narave in odvisni od aplikacije in vhodnih podatkov. **Dinamični** principi so zelo primerni za razvrščanje opravil v realnem času. Upoštevajo dinamično spreminjanje lastnosti opravil in se prilagajajo dogajanju v sistemu. Verjetnost zakasnitev odziva sistema na pomembne dogodke je nižja, kot pri statičnih principih. **Mešani** principi izkoriščajo prednost statičnih in dinamičnih principov. Pri namenskih sistemih je zelo pomembno, da je procesor prost v vsakem trenutku. Zato je stopnja izkoriščenosti procesorja nizka (do 20 %). To lastnost izkoriščajo mešani principi, ki dodeljujejo prednost opravilom z najvišjimi prioritetami in jih razvrščajo z zahtevnejšim dinamičnim principom. Čas, ko je procesor prost, pa izkoriščajo za izvajanje preprostih opravil, ki ne zahtevajo hitrega odziva in se lahko izvajajo v vrstnem redu, določenem s statičnim principom.

opravilo	T_A	T_C	T_D	*kritičnost*
τ_1	3	2	6	2
τ_2	1	2.5	7	1
τ_3	0	3	8	3

Tabela 6.2: Lastnosti treh aperiodičnih opravil

Opis delovanja principov razvrščanja bomo opremili z grafičnim prikazom obnašanja treh aperiodičnih opravil ($\tau_i, i = 1, \ldots 3$), ki so določeni s časom prihoda (T_A), izvajalnim časom (T_C), časovno mejo (T_D) ter kritičnostjo (tabela 6.2).

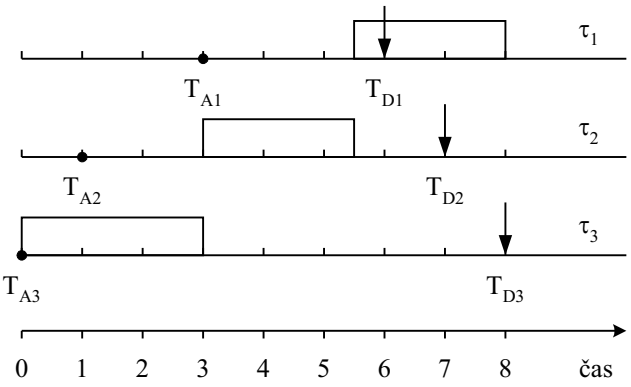
Opravilo τ_2 s stopnjo kritičnosti 1 ima najvišjo logično prioriteto.

Enostavno razvrščanje brez prekinjanja

FCFS (First-Come-First-Served)

Princip je enostaven za razumevanje in izvajanje. Razvrščevalnik razporeja opravila glede na njihov čas prihoda. Ko opravilo prične z izvajanjem, ga izvrševalnik ne sme prekiniti, če tudi sistem zahteva takojšen odziv pomembnejšega opravila (slika 6.6).

Mešani principi pogosto uporabljajo princip FCFS pri razvrščanju manj kritičnih opravil, ki se lahko izvajajo "v ozadju", ko je procesor prost in sistem ne zahteva izvajanja pomembnih opravil. Princip je razširjen, tako da dopušča prekinjanje izvajanja opravil, če sistem zahteva takojšen odziv na kritične dogodke.

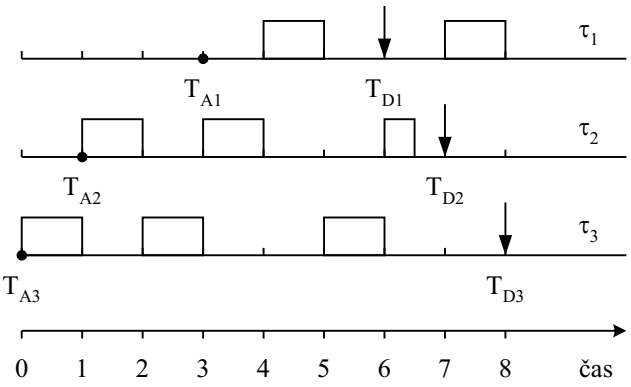


Slika 6.6: FCFS princip razvrščevanja opravil

Enostavno razvrščanje s prekinjanjem

Princip krožnega dodeljevanja (Simple Round-Robin)

Princip krožnega dodeljevanja je enostaven pristop k razvrščanju, ki podpira prekinjanje izvajanja opravil. Razvrščevalnik daje prednost opravilom, ki so se prva prebudila. Po izteku določenega časa (časovne rezine -*time slice*) se prekine izvajanje opravila in procesor prevzame naslednje opravilo. Prekinjeno opravilo nadaljuje z izvajanjem, ko vsa ostala prebujena opravila izkoristijo časovno rezino (slika 6.7).



Slika 6.7: Princip krožnega dodeljevanja

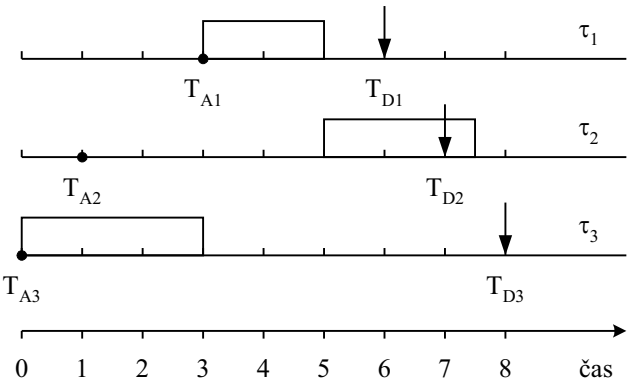
Časovna rezina je sorazmerna osnovni časovni enoti (*clock tick*). Časovna rezina ne sme biti prekratka, sicer je čas preklapljanja nesorazmerno velik v primerjavi s celotnim časom izvajanja opravil. Zaželena je dolžina časovne rezine, ki je malo večja od od povprečnega časa izvajanja opravil. Če je rezina predolga in se opravilo zaključi pred iztekom tekoče časovne rezine, mora naslednje opravilo čakati na naslednjo časovno rezino, čeprav so viri prosti. Zgornja meja časovne rezine je omejena z najslabšim odzivnim časom opravila.

Princip ni uporaben za razvrščanje opravil v realnem času. Predstavlja pa osnovo bolj učinkovitim principom razvrščanja. Dolžina časovne rezine je lahko spremenljiva in se prilagaja glede na prioritete opravil. Vrstni red izvajanja opravil se lahko dinamično spreminja ob izteku vsake časovne rezine.

Prioritetno razvrščanje brez prekinjanja

a) Princip prednosti krajših opravil (Shortest-Job-First)

Razvrščevalnik daje prednost opravilom, katerih čas izvajanja je krajši. Princip zagotavlja v primerjavi z enostavnimi principi razvrščanja krajši povprečni čakalni čas prebujenih opravil. Vendar je tak čas bolj nepredvidljiv, zlasti pri daljših opravilih (slika 6.8).



Slika 6.8: Princip prednosti krajših opravil

V času $t = 3$, ko se izvrši opravilo τ_3, sta pripravljeni dve opravili, τ_1 in τ_2. τ_1 ima krajši predvideni čas izvajanja, zato prevzame procesor.

Izvrševalnik OS za delo v realnem času uporablja princip za razvrščanje manj kritičnih in časovno zahtevnih opravil, ki se lahko ozvajajo, ko je procesor prost in sistem ne zahteva izvajanja pomembnih opravil. Princip mora dopuščati prekinjanje izvajanje opravil, če sistem zahteva nenaden odziv na dogodke.

b) Princip prednosti čakajočih opravil (Highest-Response-Ratio-Next)

Princip prednosti čakajočih opravil odpravlja nekatere slabosti *principa prednosti krajših opravil*. Pri razvrščanju upošteva poleg izvajalnih časov opravil tudi njihov čakalni čas. Ko opravilo zaključi z izvajanjem opravila, razvrščevalnik izračuna funkcijo za vsako opravilo, na osnovi katerih določi vrstni red izvajanja opravil:

$$f(T_L, T_C) = \frac{T_L + T_C}{T_C}$$

Opravilo z najvišjo vrednostjo funkcije ima prednost pri dostopu do procesorja (slika 6.9).

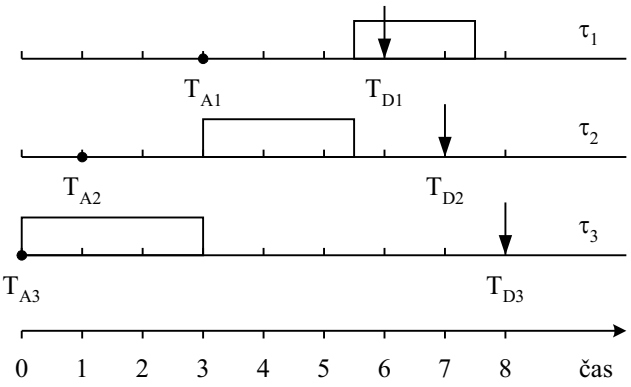
V času $t = 3$ je vrednost funkcije za čakajoče opravilo τ_1 $f_{\tau 1}(T_{L1}, T_{C1}) = (0 + 2)/2 = 1$ ter za opravilo τ_2 $f_{\tau 2}(T_{L2}, T_{C2}) = (2 + 2.5)/2.5 = 1.8$. Zato prevzame procesor opravilo τ_2, ki ima višjo vrednost funkcije.

Princip zagotavlja v primerjavi s *principom prednosti krajših opravil* boljši odzivni čas sistema. Ker ne dopušča prekinjanja opravil in zahteva dodaten čas za izračun funkcije za vsako opravilo, ga običajno ne uporabljamo v namenskih aplikacijah za delo v realnem času.

Prioritetno razvrščanje s prekinjanjem

1. Statični principi

Statični principi upoštevajo pri razvrščanju logične ali časovne prioritete prebujenih opravil. Procesor prekine izvajanje opravila le v primeru, ko sistem sproži zahtevo za izvajanje opravila z višjo prioriteto. Razvrščevalnik priredi razvrstitev opravil ob prihodu novega opravila.

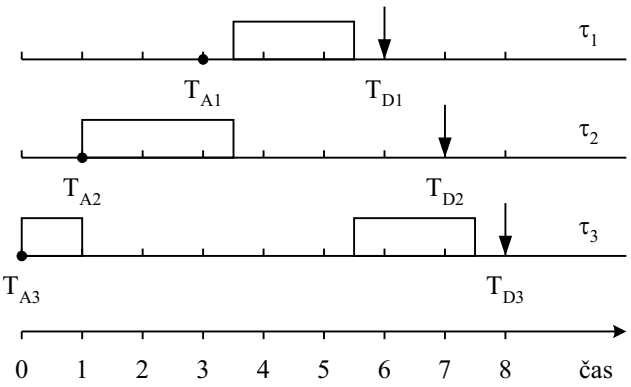


Slika 6.9: Princip prednosti čakajočih opravil

Statični principi se že dalj časa uveljavljajo pri OS za delo v realnem času. Kljub enostavni izvedbi njihova uporaba pogosto ni možna pri namenskih sistemih, saj le ti zahtevajo upoštevanje vseh lastnosti, tako časovnih, kot tudi logičnih.

a) Princip prednosti kritičnih opravil

Princip upošteva le kritičnost opravil pri določanju zaporedja izvajanja prebujenih opravil na procesorju (slika 6.10)



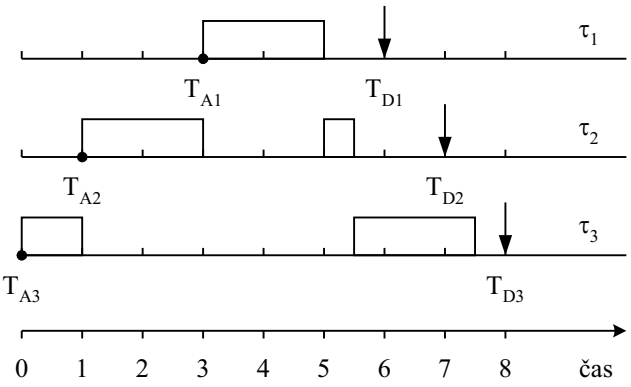
Slika 6.10: Princip prednosti kritičnih opravil

V času $t = 1$ prekine izvajanje τ_3 opravilo τ_2, ker ima višjo prioriteto zaradi višje stopnje kritičnosti. Ko zaključi z izvajanjem, dodeli razvrščevalnik prednost pri dostopu do procesorja opravilu τ_1, ker ima višjo prioriteto od čakajočega opravila τ_3.

Kritičnost opravila določa programer hevristično, zato obravnavamo princip kot subjektiven pristop k reševanju problema razvrščanja. Princip je statičen, ker se kritičnost opravila ne spreminja med izvajanjem aplikacije.

b) Princip prednosti opravil s krajšim odzivnim časom

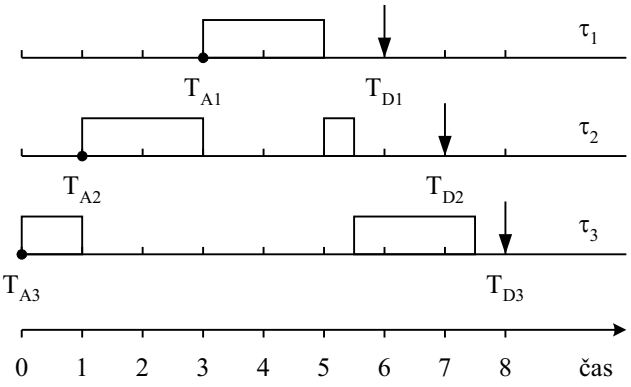
Princip daje prednost pri dodelitvi procesorja opravilom, ki imajo krajši odzivni čas (T_R) (slika 6.11). Ta časovna lastnost opravil je določena z aplikacijo namenskega sistema, zato vključimo princip med objektivne pristope.



Slika 6.11: Princip prednosti opravil s krajšim odzivnim časom

c) Princip prednosti krajših opravil

Princip daje prednost opravilom s krajšim predvidenim časom izvajanja (T_C) (slika 6.12). Predvideni čas izvajanja je ocenjen za "najslabši"primer, ko vhodni podatki narekujejo najdaljši možni čas izvajanja.



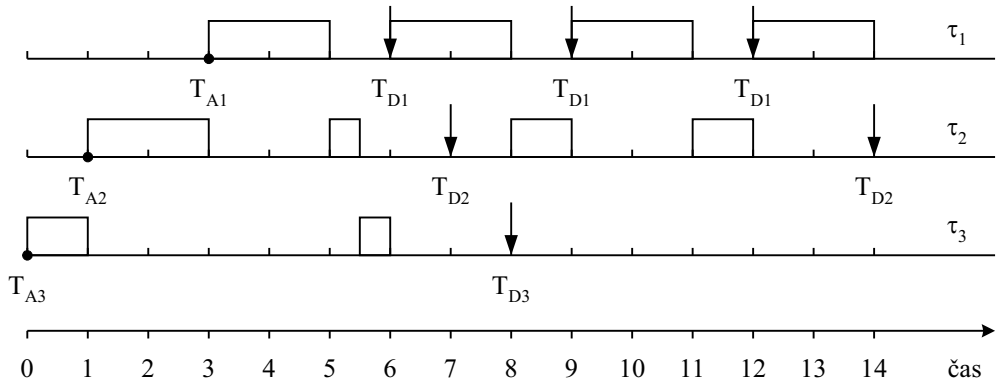
Slika 6.12: Princip prednosti krajših opravil

d) Princip prednosti pogostejših opravil (Rate Monotonic)

Zelo pogosto uporabljen princip razvrščanja pri jedrih je t.i. princip prednosti pogostejših opravil. Temelji na prepodstavki, da so opravila aplikacije periodična in sistem ne podpira asinhronih dogodkov.

Princip omogoča prednost opravilom, katerih pogostnost izvajanja je največja. Pogostnost izvajanja je obratno sorazmerna z dolžino periode ($1/T_P$). Slika 6.13 prikazuje obnašanje treh opravil, vendar s predpostavko, da se izvajajo periodično.

Dobre lastnosti principa žal temeljijo na predpostavki, da so opravila enakovredna (enako kritična za aplikacije) in medsebojno neodvisna. Problem, kateremu se razvrščevalnik z uporabo tega principa težko izogne, je t.i. *problem prioritetne inverzije (priority inversion).* Opravila, ki niso kritična za aplikacijo, lahko zaradi pogostnosti (visoke frekvence) izvajanja zadržujejo odziv kritičnih opravil z daljšimi periodami. Problemu, da se nekatera opravila ne izvršijo pravočasno, se je zelo težko izogniti, zlasti če čas prihoda opravil ni znan pred izvajanjem aplikacije. Sicer lahko programer predvidi napako zaradi prioritetne inverzije in priredi ustrezno razvrstitev opravil (priority transformation).



Slika 6.13: Princip prednosti pogostejših opravil

2. Dinamični principi

Izvajanje aplikacij v realnem času je zapleteno, saj zahtevajo namenski sistemi dinamično prilagajanje "zunanjim" dogodkom. Pomembno vlogo igra jedro s sistemskimi uslugami (*services*), ki močno olajšajo delo programerju. Dinamični principi razvrščanja pomenijo eno takšnih uslug, saj obravnavajo lastnost opravil dinamično.

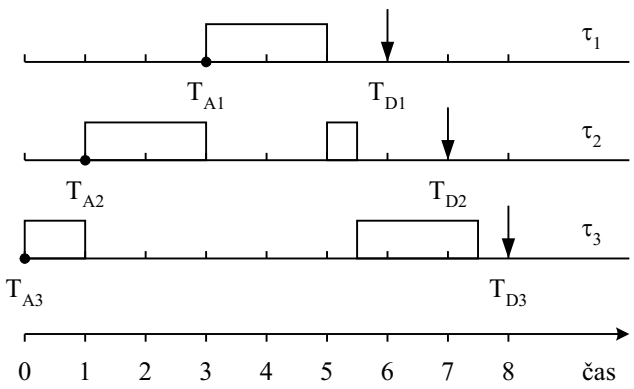
Podali bomo *princip prednosti časovno kritičnih opravil (Earliest Deadline)*, ki se uveljavlja v dveh različicah. Njegova pomanjkljivost je v prepodstavki, da so opravila mesebojno neodvisna in enakovredna. Pri razvrščanju upošteva le časovne lastnosti opravil.

Princip prednosti časovno kritičnih opravil

Asinhronost delovanja sistema za delo v realnem času je lastnost, ki jo mora razvrščevalni princip upoštevati, sicer je njegova uporaba omejena na skromen nabor namenskih aplikacij.

Princip prednosti časovno kritičnih opravil se uveljavlja v dveh različicah, ki omogočata izvajanje opravila v *prvem (Earliest Deadline As Soon As Possible)* ali *zadnjem možnem trenutku (Earliest Deadline As Late As Possible)*.

Prva različica principa dodeljuje prednost tistim opravilom, katerih čas T_D je bližje trenutnemu času t (slika 6.14). V času $t = 3$ je časovna meja T_{D_1} opravila τ_1 najbližje času t, zato nadaljuje τ_1 z izvajanjem. Pravimo da je opravilo τ_1 v trenutku $t = 3$ časovno najbolj kritično.



Slika 6.14: Princip prednosti časovno kritičnih opravil - prva različica

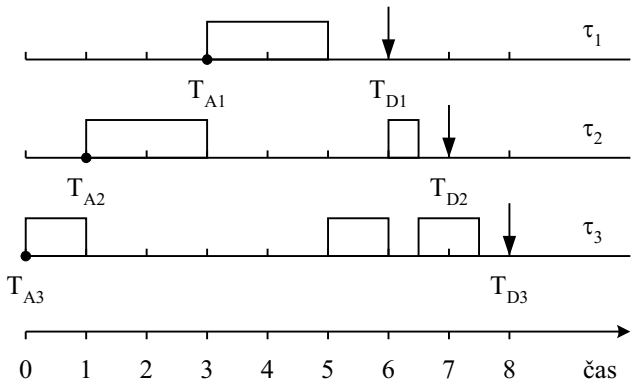
Druga različica je časovno bolj potratna, ker zahteva v vsakem časovnem trenutku t izračun funkcije:

$$slack(\tau_i, t) = max(T_{L_i}, 0) \tag{6.4}$$

za vsako prebujeno opravilo τ_i. Čas T_{L_i} je določen s časom T_{D_i} in s predvidenim časom izvajanja T_{C_i}. Prednost pri dostopu do procesorja ima opravilo, katerega funkcija ima najmanjšo vrednost (slika 6.15). Opravilo τ_i z negativnim časom T_{L_i} ($T_{D_i} - T_{C_i} < t$) je zavrnjeno, saj se gotovo ne bo izvršilo pravočasno. V takšnem primeru vrne funkcija vrednost 0.

V času $t = 5$ ima opravilo τ_2 vrednost funkcije $T_{L_2} = T_{D_2} - t - T_{C_2}(t) = 7 - 5 - 0.5 = 1.5$ ter τ_3 $T_{L_3} = 8 - 5 - 2 = 1$. Ker ima τ_3 nižjo vrednost funkcije, nadaljuje z izvajanjem na procesorju.

Matematični in simulacijski model, ki sta ga podala Craig in Woodiside, podaja primerjavo učinkovitosti opisanih različic principa. Verjetnost prekoračitve časovne omejenosti opravil je pri slednji različici večja kot pri prvi, ki omogoča izvajanje opravila, takoj ko dobi le-to najvišjo prioriteto (postane časovno najbolj kritično).



Slika 6.15: Princip prednosti časovno kritičnih opravil - druga različica

6.1.2 Principi razvrščanja pri večprocesorskih in porazdeljenih sistemih

Namenske aplikacije, ki se izvajajo v realnem času, postajajo vse bolj zahtevne. Število opravil se povečuje, kot tudi kompleksnost komunikacije z okoljem. Enoprocesorske računalniške sisteme nadomeščajo zmogljivejši, večprocesorski in porazdeljeni sistemi. Izvajanje aplikacij v takšnih sistemih se pohitri, saj povečano število procesorjev in ostalih virov omogoča *vzporedno* in ne le sočasno izvajanje opravil. Seveda se hkrati poveča tudi zahtevnost nadzora nad obnašanjem aplikacij.

Večprocesorske sisteme imenujemo *tesno sklopljeni (tightly-coupled)* in porazdeljene *šibko sklopljeni (loosely-coupled)* sistemi. Arhitektura sistemov se razlikuje, zato bomo podali principe razvrščanja opravil ločeno za obe skupini.

Tesno sklopljeni sistemi združujejo več procesorskih enot, ki so povezane preko skupnega naslovnega in podatkovnega vodila. Vse enote imajo dostop do skupnega pomnilnika, preko katerega poteka izmenjava podatkov med opravili in njihova sinhronizacija. Vsaka enota lahko uporablja dodatni lokalni pomnilnik za shranjevanje internih podatkov.

Šibko sklopljeni sistemi združujejo več računalniških sistemov, ki so povezani s komunikacijskimi povezavami.

Naloga izvrševalnika je zagotoviti pravočasen in zanesljiv odziv na dogodke v sistemu. Ker so vhodno/izhodne povezave računalniškega sistema z okoljem porazdeljene, je zaznava dogodkov in izvajanje ustreznih opravil otežkočeno zlasti v porazdeljenih sistemih. Nekateri sistemi izvajajo nadzor s pomočjo jedra, ki centralno upravlja z dogodki in porazdeljuje opravila po procesorjih. Druga rešitev pa temelji na porazdeljenem nadzoru posameznih procesorskih ali računalniških enot, ki jih izvaja več "kopij" jedra, neodvisno na vsaki enoti.

V tesno sklopljenih sistemih sta oba pristopa učinkovita. Medtem, ko "centralni" pristop zahteva več prostora v skupnem pomnilniku, lahko nastopijo pri "porazdeljenem" pristopu težave pri komunikaciji med opravili. Kopije jedra so shranjene v lokalnih pomnilnikih procesorskih enot, zato je dostop do sistemskih opravil zelo hiter.

Šibko sklopljenih sistemov ne moremo nadzirati s centralnim jedrom, saj nimajo skupnega pomnilnika. Lahko pa izberemo računalniško enoto, ki izvaja le sistemska opravila jedra. Ta, t.i. aparaturna rešitev je zelo učinkovita, saj razbremeni ostale enote, ki lahko posvetijo celoten procesorski čas izvajanju aplikacije. Večina šibko sklopljenih sistemov uporablja "porazdeljeni"nadzor.

Centralni nadzor

Če so procesorske enote enakovredne, je problem večprocesorskega razvrščanja opravil olajšan. Razvrščevalnik porazdeli prebujena opravila enakomerno po sistemu. V literaturi lahko zasledimo številne optimizacijske metode za porazdeljevanje opravil po večih procesorjih.

Pri aplikacijah, katerih opravila so zahtevnejša, lahko opravimo razbitje operacij opravil, ki jih izvajamo vzporedno na večih procesorjih. Pristop je zelo zahteven, zlasti, če zahtevamo prekinjanje izvajanja opravil.

Omenjena pristopa sta učinkovita, če so opravila periodična in medsebojno neodvisna. Zelo pogost pristop, ki ga uporabljajo oblikovalci programske opreme namenskih sistemov za delo v realnem času, je t.i. *uravnoteženo porazdeljevanje*. Pristop upošteva naključnost dogodkov in medsebojno odvisnost opravil. Uravnoteženo porazdeljevanje upošteva možnost nenadnega nasičenja sistema z nujnimi dogodki. Razvrščevalnik razbremeni procesor, ki ne zmore opraviti vseh opravil, s posredovanjem zahtev za izvajanje opravil sosednjim procesorskim ali računalniškim enotam. Pristop je zamuden, saj zahteva dodaten čas za vzpostavitev zveze med jedrom in enotami. Jedro hrani med izvajanjem aplikacije podatke o možnih povezavah med enotami. V literaturi zasledimo številne algoritme za iskanje najbolj ugodnih zvez.

Namenski večprocesorski sistemi imajo dodatno aparaturno omejitev, ki je do sedaj opisani principi ne upoštevajo. Povezava računalniškega sistema z okoljem je običajno porazdeljena po sistemu. Vhodno/izhodne naprave so priključene na različne procesorske enote. Centralni nadzor je tako otežkočen in zelo težko izvedljiv.

Porazdeljeni nadzor

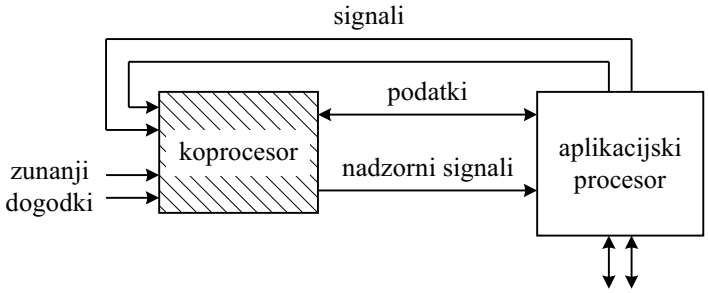
Nadzor, ki ga izvaja ločeno več kopij jedra, rešuje omenjeni problem centralnih pristopov. Pojavijo pa se nove ovire, ki nastopijo ob nasičenju zahtev za izvajanje opravil. Procesor, ki ne more izvesti vseh zahtevanih opravil, razbremeni vire s posredovanjem nekaterih opravil drugim procesorskim enotam. Predhodno mora izvrševalnik "nasičenega"procesorja vzpostaviti zvezo z ostalimi kopijami jedra. To pa zahteva dodaten čas in hkrati tudi povečanje možnosti napake pri prenosu podatkov med procesorskimi enotami. Pristop je razširitev centralnega pristopa *uravnoteženega porazdeljevanja* opravil.

"Sistemski" koprocesor

Kot je razvidno iz kratke predstavitve principov razvrščanja opravil v večprocesorskih sistemih, odločitev, kateri pristop je bolj učinkovit, ni enostavna. Halang je pokazal, da je centralni pristop porazdeljevanja zelo učinkovit, če se izvajajo sistemska opravila na ločenem *(ko)procesorju*.

Koprocesor zaznava dogodke (prekinitve) in odloča, katera opravila imajo prednost pri izvajanju. Procesorji, ki izvajajo aplikacijska opravila, so povezani preko vhodno/izhodnih vmesnikov z okoljem (slika 6.16).

Koprocesor deluje vzporedno z ostalimi procesorji in omogoča njihovo delovanje brez nepotrebnih prekinjanj. Tako se poveča zanesljivost sistema, saj je odziv na dogodke takojšen, prekinitve pa niso onemogočene zaradi morebitnega izvajanja opravil na skupnih virih.



Slika 6.16: Koprocesor izvaja sistemska opravila

Pristop z uporabo koprocesorja je učinkovit tudi v enoprocesorskih namenskih sistemih. Halang **[literatura]** opisuje razvoj VLSI koprocesorske enote, ki izvaja sistemsko upravljanje z opravili. Sistemska opravila koprocesorja temeljijo na Adinem modelu upravljanja z opravili. Koprocesor nudi 27 operacij, katerih hitrost izvajanja je v primerjavi s programsko izvedbo 100-krat večja.

6.2 Komunikacija med opravili

Predstavimo problem neposredne odvisnosti opravil, ki je pogojena s časovnimi in logičnimi dogodki v sistemu. Opisali bomo tehnike, ki podpirajo komunikacijo med opravili aplikacije:

- sinhronizacija,
- prenos podatkov ter
- prenos podatkov s sinhronizacijo.

Medtem, ko zaščita skupnih virov povzroča medsebojno izključevanje opravil, omogoča komunikacija njihovo sodelovanje in obveščanje o dogodkih, ki vplivajo na nadaljni potek izvajanja opravil.

Sinhronizacija

Sinhronizacija zagotavlja časovno usklajenost izvajanja operacij opravil. Tehnika temelji na uporabi skupnih spremenljivk (signalov), katere smo predstavili pri opisu upravljanja skupnih virov.

Običajno podpira izvrševalnik sinhronizacijo opravil s tremi tipi signalov, *send*, *wait* in *check*. Opravilo, katere operacije zahtevajo sinhronizacijo z ostalimi opravili, pošlje signal tipa *send*. Izvrševalnik prekine izvajanje tega opravila, če nobeno opravilo ne pričakuje tega signala. Podobno se prekine izvajanje opravila, ki je poslalo signal tipa *wait*, dokler se ne pojavi signal tipa *send* (slika 6.17).

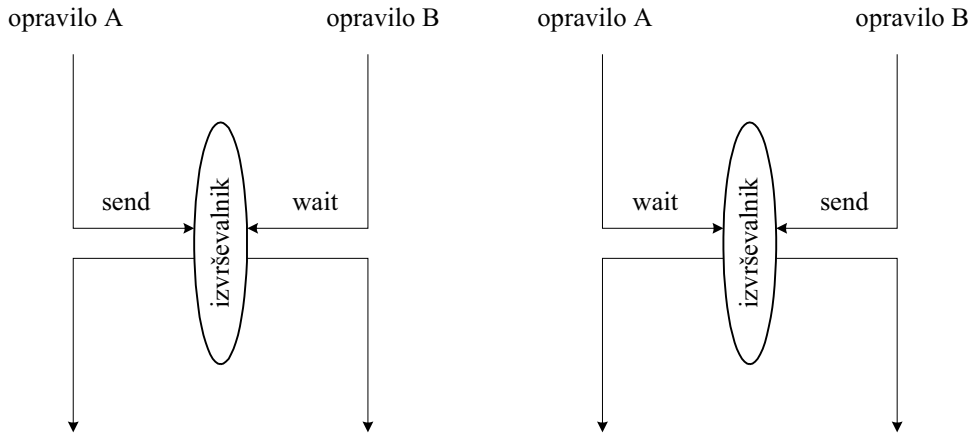
Sistemsko opravilo izvrševalnika, ki preverja prisotnost signalov tipa *send* ali *wait*, sporoča stanje s signali tipa *check*. Tako lahko opravilo pošlje signal tipa *send*, ko je le-ta zagotovo pričakovan. Signal tipa *check* preprečuje problem mrtve zanke, ki se pojavi, ko opravilo čaka na signal čakajočega opravila.

Pošiljanje signalov ne poteka neposredno med opravili aplikacije, temveč posredno pod nadzorom izvrševalnika.

Prenos podatkov

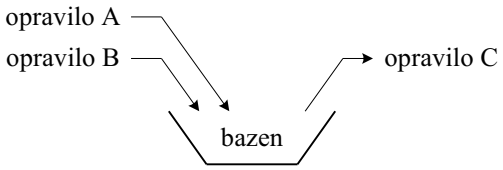
Opravila aplikacije lahko zahtevajo izmenjavo podatkov. Izvrševalnik mora zagotoviti pravočasen in zanesljiv prenos podatkov med opravili, saj napake pri prenosu podatkov lahko porušijo pravilno delovanje aplikacije.

Zahteve po prenosu podatkov med opravili se lahko pojavljajo periodično ali povsem naključno (asinhrono). Mehanizma, ki omogočata prenos podatkov brez sinhronizacije opravil, sta t.i. *shranjevanje podatkov v bazene* in *kanale*.



Slika 6.17: Sinhronizacija opravil

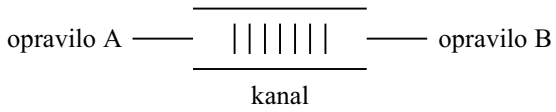
Pri prvem mehanizmu določa izvrševalnik del pomnilnika RAM kot bazen za hranjenje podatkov, ki so skupni opravilom (npr. sistemske tabele, vrednosti koeficientov). Opravila imajo možnost branja teh podatkov, ne morejo pa jim spreminjati vrednosti (slika 6.18).



Slika 6.18: Bazen - struktura za shranjevanje podatkov

Bazen je skupni vir, zato mora izvrševalnik zagotoviti varen dostop do podatkov v bazenu s pomočjo metod upravljanja skupnih virov.

Kanal je podatkovna struktura za shranjevanje skupnih podatkov, do katerih imata dostop le dva procesa (slika 6.19). Za razliko od bazenov, se podatek pri branju izloči iz kanala. Ker je izvedba "cevovodna", lahko opravili vpisujeta in bereta podatke iz kanala asinhrono. Običajno so podatki, oz. kazalci na podatke, urejeni v kanalu po principu FIFO (First-In-First-Out). Cevovodna izvedba kanala lahko uporablja podatkovne strukture, kot sta npr. *vrsta* ali *krožni vmesnik*. Dolžina vrste je omejena le z velikostjo razpoložljivega pomnilnika. Vendar pa ni zaželena uporaba predolgih vrst zaradi zakasnitev, ki se lahko pojavijo med vpisom in branjem podatkov.



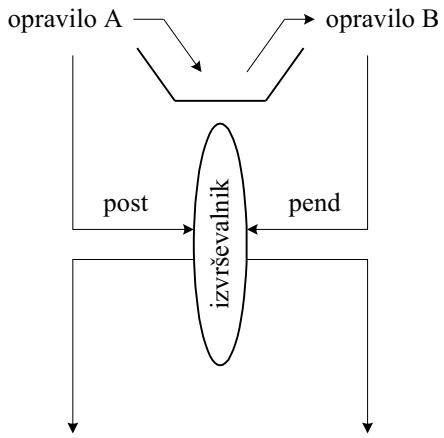
Slika 6.19: Kanal - struktura za shranjevanje podatkov

Velikost krožnega vmesnika je določena enkratno in se med izvajanjem aplikacije ne spreminja.

Prenos podatkov s sinhronizacijo

Med izvajanjem opravil aplikacije se lahko pojavi zahteva po prenosu podatkov ob določenem dogodku. Mehanizem *poštnega nabiralnika* omogoča sinhronizacijo izvajanja opravil (s pomočjo signalov) ter dostop do podatkovne strukture za shranjevanje podatkov.

Dostop do poštnega nabiralnika ima lahko več opravil hkrati. Opravilo, ki zahteva prenos podatkov, pošlje signal (*post*) izvrševalniku ter shrani podatke v podatkovno strukturo nabiralnika (npr. bazen). Izvajanje opravila se prekine (izvrševalnik pošlje signal tipa *wait*), dokler opravilo, ki je vložilo zahtevo za sprejem teh podatkov (s pomočjo signala *pend*), ne prebere vsebine nabiralnika. Podobno se ustavi izvajanje opravila, ki je poslalo signal *pend*, dokler podatki niso dostavljeni v nabiralnik in izvrševalnik ne sprejme signala *post* (slika 6.20).



Slika 6.20: Princip poštnega nabiralnika

Velikost podatkovne strukture nabiralnika ni omejena, saj le-ta hrani kazalce na podatke.

6.3 Primer OS za delo v realnem času

Kot primer učinkovitega OS za delo v realnem času predstavimo VRTX32. Številni namenski sistemi delujejo pod nadzorom VRTX32 OS. Ker omogoča visoko stopnjo zanesljivosti, so ga izbrali kot nadzornika elektronskega sistema za letalsko vodenje in upravljanje v potniškem letalu McDonnell Douglas MD-11.

VRTX32 se odlikuje z modularno strukturo, ki omogoča izbor potrebnih sistemskih uslug glede na zahtevnost namenskega sistema. VRTX32 predstavlja vmesnik med programsko aplikacijo in strojno opremo namenskega sistema. Hkrati podpira tudi razvoj v večjih programskih jezikih in različnih razvojnih okoljih. Prilagojen je za uporabo v mikroračunalnikih, kot tudi v transputerskih sistemih, kot je npr. T9000 transputer.

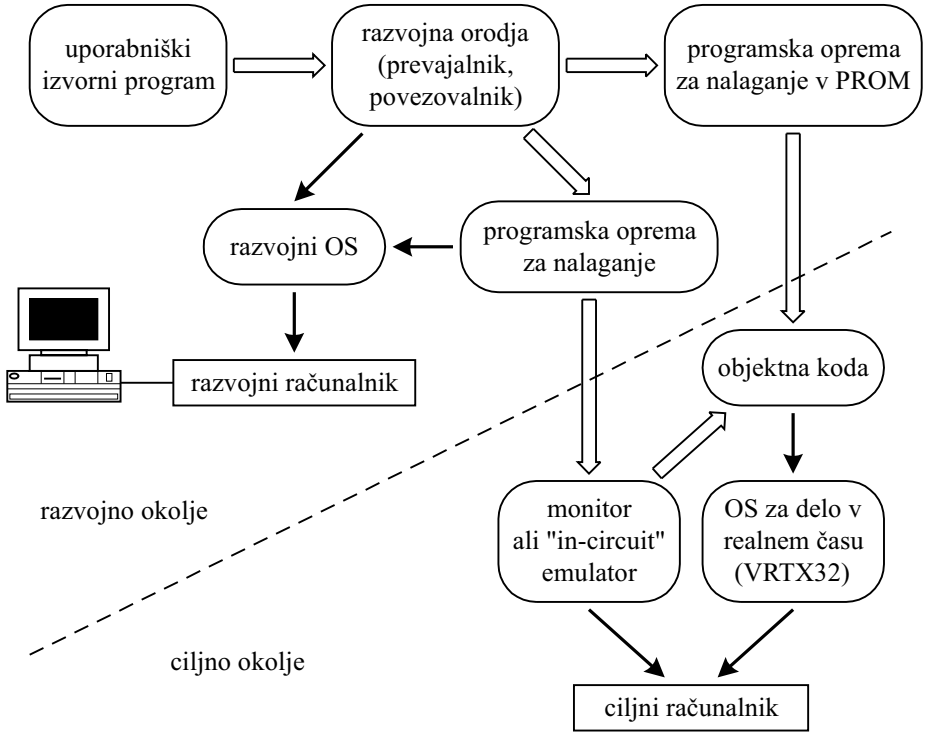
Poleg funkcij jedra nudi podporo za delo z vhodno/izhodnimi napravami, datotekami, multiprocesiranje skupnega pomnilnika in porazdeljeno procesiranje v lokalnih mrežah. Velikost jedra je le 9 KB, celotni OS pa zahteva pomnilniški prostor 100 KB.

Jedro omogoča razvrščanje po *principu prednosti kritičnih opravil*. Komunikacija med opravili poteka preko *poštnih nabiralnikov*. VRTX32 podpira dinamično dodeljevanje pomnilnika v blokih nespremenljivih dolžin. Zaradi večje izkoriščenosti prostora omogoča razbitje pomnilnika na particije, ki vsebujejo bloke različnih dolžin. Vsaka particija združuje bloke iste dolžine. Tako lahko opravilo izbira različno dolge bloke. VRTX32 vsebuje sistemske procedure, ki omogočajo izmenjavo podatkov med opravili in vhodno/izhodnimi napravami. Prav tako nudi možnost uporabe prekinitvenih podprogramov.

Pomemben podatek je *hitrost preklopa* med opravili, ki znaša 20 do 80 μs (Motorola 68020, 25MHz). Prekinitve so onemogočene povprečno 10 μs.

Slika 6.21 prikazuje postopek razvoja namenske aplikacije.

VRTX32 podpira prenos strojnih ukazov v ROM pomnilnik ciljnega (*target*) računalnika (silicijev OS). Takšen pristop omogoča izvajanje aplikacij neodvisno od pomnilniške lokacije, ki jo zasede ROM



Slika 6.21: Razvoj namenskih aplikacij

čip v sistemu.

Namenske aplikacije, katerih strukturo narekuje VRTX32, lahko oblikujemo s pomočjo CASE (Computer Aided Software Engineering) orodij, kot je npr. VRTXdesigner. Orodje omogoča preverjanje odvisnosti med opravili ter simulacijo razvrščanja.

Poglavje 7

Arhitekture signalnih procesorjev

7.1 Uvod

7.1.1 Kaj je DSP?

DSP (Digital Signal Processor) je namenski procesor namenjen digitalni obdelavi signalov. Na vhodu DSP-ja je A/D (analogno / digitalni) konverter, ki časovno zvezni signal pretvori v časovno diskretnega. To pomeni, da dobi DSP na svojem vhodu (ali več vhodih) vzorčeni signal v obliki $x_i(n)$, ga obdela in da izhode v obliki $y_i(n)$. Izhodni signali DSP-ja gredo dalje v D/A (digitalno/analogni) konverter, ki jih pretvori nazaj v zvezni signal.

DSP-ji delajo z velikim številom vzorcev na sekundo, zato zahtevajo velike prenose podatkov in veliko procesorske moči. Taka obdelava v realnem času pa je prevelika obremenitev za konvencionalne procesorje.

Poznamo programirljive in namenske DSP-je. Programirljivi procesorji so bolj fleksibilni od namenskih, ki so že hardversko programirani za določen algoritem. So pa zato namenski procesorji načeloma hitrejši in porabijo manj moči.

Glavne funkcije, ki jih opravljajo DSP-ji so filtriranje in Fourierova transformacija.

7.1.2 Parametri DSP-ja

Čas med dvema vzorcema (sample period) pogojuje kvaliteto oziroma frekvenčni obseg digitaliziranega signala. Shannonov teorem pravi, da mora biti vzorčna frekvenca vsaj dvakrat večja od največje frekvence, ki jo vzorčimo. Vzorčna frekvenca je torej eden od parametrov za ocenjevanje DSP-ja. Drugi pomemeben parameter je zamik zaradi računanja (computational latency). Vsak izhodni vzorec ustreza nekemu vhodnemu. Zaradi obdelave vhodnih vzorcev pa pride do časovnega zamika.

Druga dva parametra, ki sta še pomembna sta velikost integriranega vezja (VLSI implementation area) in poraba moči (power dissipation).

7.1.3 Primerjava z mikroprocesorji

Mikroprocesorji in DSP-ji imajo precej skupnih lastnosti, vendar obstajajo med njimi pomembne razlike. DSP-ji so primarno načrtovani za procesiranje podatkov v realnem času. Imajo optimizirane V/I (vhodno/izhodno) enote, dvojni pomnilnik (za podatke in za ukaze), množilno enoto (zelo pomembno, saj s tem nadomestimo veliko število seštevanj) in več drugih lastnosti, ki pripomorejo k veliki moči teh procesorjev.

Prednosti DSP-jev pred mikroprocesorji :

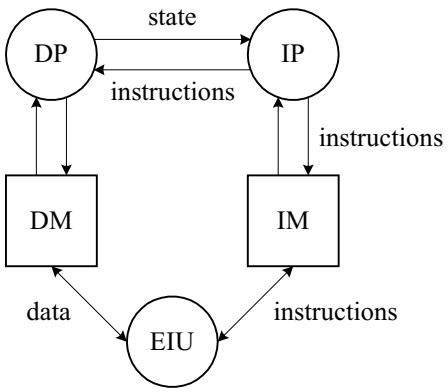
- zaradi specifične namenskosti so cenovno zelo ugodni
- dolžino besede lahko izbiramo glede na aplikacijo (pri mikroprocesorjih je fiksna)

- A/D in D/A pretvornika sta po navadi že vgrajena v sam DSP (povečana fleksibilnost)
- zaradi majhne porabe so zanimivi za prenosne aplikacije
- zaradi obstoja velikih knjižnic z optimizirano kodo za DSP- je je programiranje lažje (v višjenivojskih jezikih), kar zniža ceno
- enostavna povezava več DSP-jev v večprocesorske sisteme.

7.2 Klasifikacija arhitektur

7.2.1 Abstraktni model

Abstraktni model DSP-ja je prikazan na sliki 7.1.



Slika 7.1:

Sestavljen je iz več funkcionalnih enot. Ukazni procesor IP (Instruction Processor) je enota, ki skrbi za interpretacijo ukazov, ki jih predaja podatkovnemu procesorju DP-ju (Data Processor), kateri jih potem izvršuje nad podatki. Vsaka od teh enot ima svoj spomin. DM (Data Memory) hrani operande, ki jih bo uporabljal DP, medtem ko IM (Instruction Memory) hrani ukaze za IP. Enota EIU (External Interface Unit) pa nadzira dostop do zunanjih podatkov, ukazov ali do drugih zunanjih naprav. Načeloma je EIU zelo močna enota, saj ima neposreden dostop do obeh pomnilnikov (DM in IM) ne ozirajoč se na DP ali IP.

7.2.2 Optimizacija

Delovanje DSP-jev lahko optimiramo na več načinov :

Transformacija diagrama stanj

Bistvo te transformacije je v prestrukturiranju diagrama stanj tako, da pospešimo čas izvajanja nekega programskega niza (cycle time). V tem primeru lahko dve stanji, ki sta med seboj neodvisni izvajamo hkrati in tako pridobimo na času.

Cevovodi (pipelines)

Cevovodi so ena najpomembnejših metod za izboljšanje zmogljivosti enoprocesorskih sistemov. S pomočjo cevovodov dosežemo istočasno (concurrent) izvajanje več ukazov tako, da se posamezni koraki prekrivajo.

Izvrševanje ukaza se razbije na manjše podukaze, za katere je potreben le majhen del celotnega časa za izvršitev ukaza. Vsakega od teh podukazov opravi točno določen del cevovoda, ki mu pravimo stopnja cevovoda. Te stopnje so povezane zaporedno tako, da tvorijo cev: na eni strani ukazi vstopajo, se v cevovodu obdelujejo in na drugi strani izstopajo. Pomik iz ene stopnje v drugo se izvrši pri vseh istočasno. Presledek med dvema pomikoma je običajno enak eni urini periodi, ta je pa določena z najpočasnejšo stopnjo cevovoda.

Pri idealno uravnoteženem cevovodu z n stopnjami bi bila zmogljivost procesorja n-krat večja. Dejansko cevovodi niso nikoli idealno uravnoteženi. V celoti gledano se povečanje števila ukazov, ki se izvršijo v določeni periodi zgodi zaradi dveh vzrokov:

1. Manjše število urinih period na ukaz (CPI): to število je načeloma n-krat manjše, čeprav se trajanje izvrševanja enega ukaza ne spremeni
2. Krajša urina perioda : ta vpliv je ponavadi manjši. Bistvo je v tem, da so podukazi dovolj enostavni in jih je mogoče opraviti v krajšem času

Prednost cevovoda pred ostalimi načini za vzporedno procesiranje je v nevidnosti za programerje. Tudi arhitektura računalnika okoli takega procesorja ostane enaka, kar ne velja za druge vrste paralelnega izvrševanja ukazov.

Povečanje števila funkcinalnih enot

Večje število DP-jev lahko neodvisno izvaja veliko število operacij ob istem IP-ju, kar lahko močno poveča zmogljivost DSP- ja. To bistveno pripomore k izboljšanju zmogljivosti.

7.2.3 Povezave med funkcionalnimi enotami

David Skillicorn je leta 1988 razdelil povezave na štiri tipe :

- 1-to-1 ena funkcionalna enota je povezana z še eno funkcionalno enoto
- n-to-n i-ta funkcionalna enota enega tipa je povezana z i-to funkcionalno enoto drugega tipa
- 1-to-n ena funckionalna enota enega tipa je povezana z n enotami drugega tipa
- n-by-n vsaka enota enega tipa je povezana z vsemi enotami drugega tipa

7.2.4 Večnivojska klasifikacija arhitektur Signalnih Procesorjev

Klasifikacija arhitektur je izvedena hierarhično, v več nivojih. To vrsto klasifikacije je prvič predlagal David Skillicorn leta 1988.

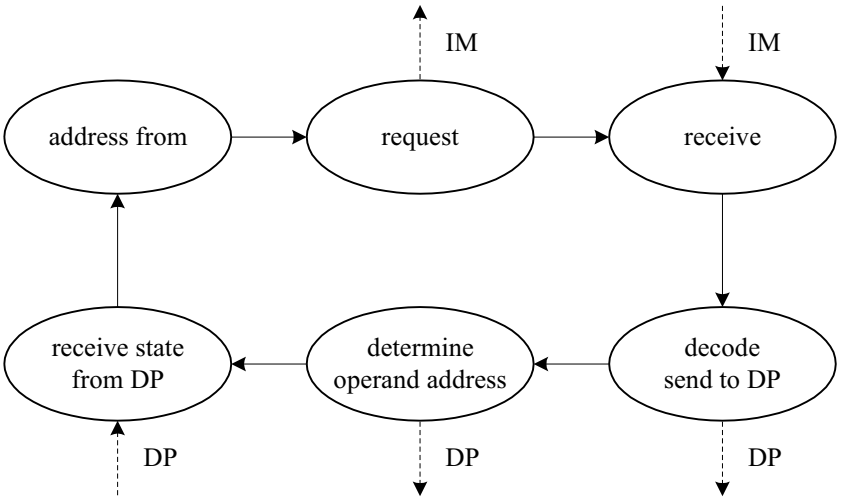
- V prvem nivoju klasifikacije je arhitektura DSP-jev opisana z navedbo števila IP-jev, IM-jev, DP-jev, DM-jev, EIU-jev in vrsto povezav med njimi.
- V drugem nivoju je klasifikacija dodatno definirana s specifikacijo diagrama stanj za vsako funkcionalno enoto posebej.

7.3 Funkcionalne enote

7.3.1 Ukazni procesor (IP)

Diagram stanj IP-ja je prikazan na sliki 7.2.

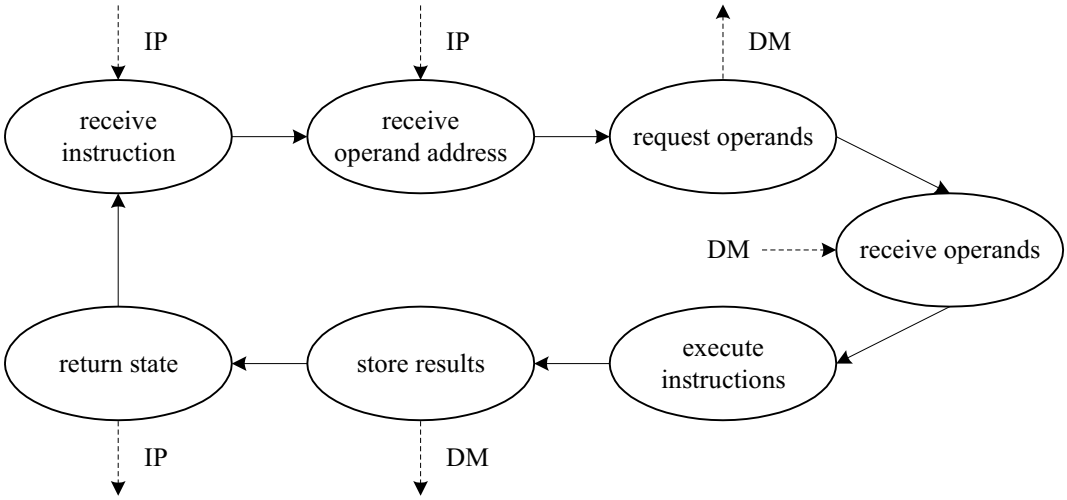
Ta najprej vzame iz programskega števca (PC - Program Counter) naslov ukaza, ki ga mora izvršiti. Nato vzame ukaz iz IM-ja, ga dekodira in ga pošlje DP-ju. Zatem določi naslove operandov, ki jih pošlje DP-ju. Nato čaka, da mu DP sporoči, da je končal.



Slika 7.2: Diagram stanj ukaznega procesorja

7.3.2 Podatkovni procesor (DP)

Diagram stanj DP-ja je prikazan na sliki 7.3.



Slika 7.3: Diagram stanj podatkovnega procesorja

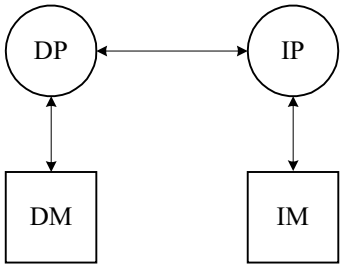
Ko DP prejme ukaz in naslove operandov od IP-ja, pošlje DM-ju zahtevo za vrednosti operandov (to lahko stori paralelno, če je na voljo več DM-jev). Nato izvrši aritmetične operacije, shrani rezultate v DM in IP-ju sporoči, da je končal.

7.3.3 DM in IM

Osnovne arhitekture

Diagram je prikazan na sliki 7.4.

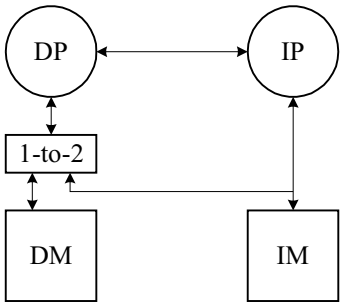
Ta osnovna arhitektura je sestavljena iz ene IM in ene DM.



Slika 7.4: Osnovna arhitektura SP

Modifikacija 1

Diagram je prikazan na sliki 7.5.

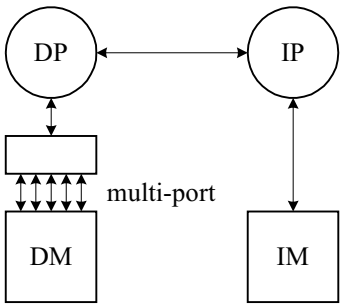


Slika 7.5: Modifikacija 1

Prva modifikacija nam omogoča shranjevanje podatkov tudi v IM, čeprav včasih ne moremo obdelovati ukazov ter podatkov hkrati. Zaradi tega pri dvooperandnih ukazih potrebujemo daljši (dva cikla) čas dostopa do pomnilnika.

Modifikacija 2

Diagram je prikazan na sliki 7.6.

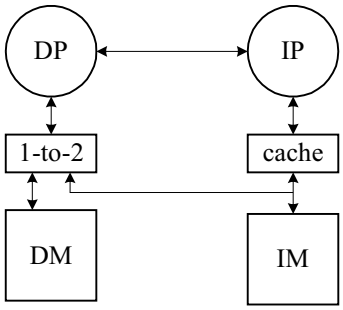


Slika 7.6: Modifikacija 2

Posebna zgradba omogoča večkraten dostop do pomnilnika znotraj ukaznega cikla, kar pospeši delovanje, hkrati pa nam zelo poveča stroške konstrukcije. Pri pomnilnikih, ki niso vgrajeni kar na čipu (off-chip memory) taka konstrukcija ni mogoča, saj smo omejeni zaradi končnega števila nogic pomnilniških čipov. To pa povzroči, da se večoperandni ukazi izvajajo več ukaznih ciklov.

Modifikacija 3

Diagram je prikazan na sliki 7.7.

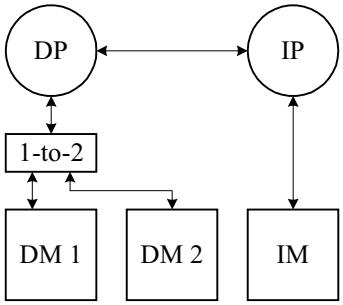


Slika 7.7: Modifikacija 3

Pri modifikaciji 1 lahko pride do konflikta med podatkovnim in ukaznim dostopom, zato dodamo predpomnilnik (cache), ki shranjuje najbolj pogosto uporabljane ukaze. S tem sprosti IM/DM pomnilnik, tako da v enem ukaznem ciklu lahko dostopamo do podatkov v tem pomnilniku.

Modifikacija 4

Diagram je prikazan na sliki 7.8.



Slika 7.8: Modifikacija 4

Namesto realizacije s predpomnilnikom lahko uporabimo dva DM namesto enega. Ta modifikacija omogoča dostop do dveh operandov in enega ukaza v enem ukaznem ciklu.

Modifikacija 5

Diagram je prikazan na sliki 7.9.

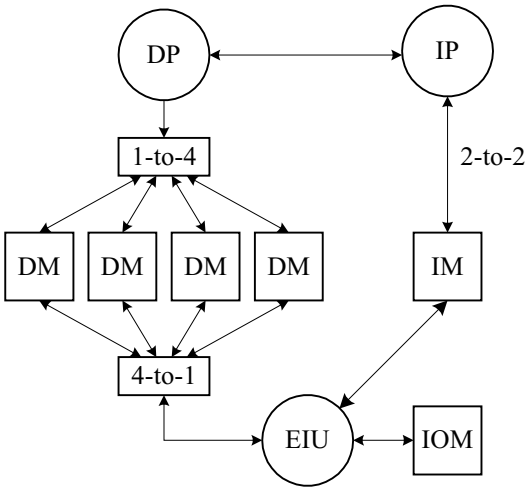
Pri tej modifikaciji imamo štiri DM-je in še dodaten pomnilnik za V/I enote. To omogoča hkratno izvajanje več operandnih ukazov skupaj z dostopi do V/I enot. Vendar pa mora programer dobro razdeliti podatke po različnih pomnilnikih, da bi dobil maksimalni izkoristek take modifikacije.

7.3.4 Zunanji vmesnik EIU (external interface unit)

EIU enote omogočajo lahek dostop do zunanjega pomnilnika (off-chip), pri vzporednem procesiranju pa kombinira izvore podatkov med posameznimi DSP-ji skupaj s skupnim pomnilnikom ter vodili tako, da dobimo zelo močno procesiranje v realnem času.

EIU enote lahko delujejo kot :

- Komunikacijska vrata



Slika 7.9: Modifikacija 5

- DMA nadzor
- Enota za arbitražo vodila
- Serijske/vzporedne V/I enote

Komunikacijska vrata

Komunikacijska vrata so sestavljena iz vhodnega in izhodnega medpomnilnika (input, output buffer), enote za arbitražo vrat (port arbitration unit - PAU) in nabora kontrolnih registrov (set of control registers). Vhodni medpomnilnik prejme sporočilo od zunanjih funkcionalnih enot medtem ko izhodni medpomnilnik vsebuje podatke, ki naj bi bili poslani drugim funkcionalnim enotam pod nadzorom PAU. PAU-i komunicirajo med seboj, da določijo način in čas prenosa podatkov med funkcionalnimi enotami. Za olajšanje uporabe protokola je na razpolago še nekaj kontrolnih in prekinitvenih signalov.

Procesor oziroma DMA nadzorna enota vpiše podatke, ki jih želi poslati drugemu procesorju v izhodni medpomnilnik (če je kaj prostora v njem). Izhodni medpomnilnik nato sporoči PAU, da želi poslati podatke. PAU lahko podatke pošlje le, če ima žeton, ki mu to dovoljuje. Če nima žetona, mora najprej zahtevati dostop do žetona. Procesor oziroma DMA kontroler nato lahko bere iz vhodnega medpomnilnika.

DMA nadzorne enote (DMA koprocesorji)

Pri DSP-jih so DMA (direct memory acces) nadzorne enote vgrajene kar na čipu z neodvisnimi podatkovnimi in naslovnimi vodili.

Ti izvajajo naslednje operacije :

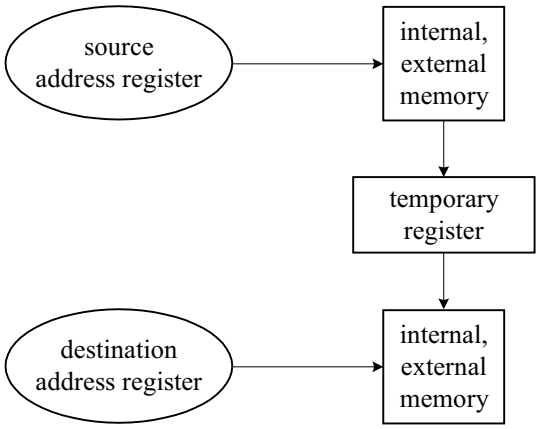
- prenos podatkov med pomnilniki (notranjimi in zunanjimi)
- prenos podatkov iz V/I enote v pomnilnik
- prenos podatkov iz pomnilnika v V/I enoto
- prenos podatkov od oziroma do komunikacijskih vrat in pomnilnikov

DMA-jev algoritem za prenos niza podatkov iz ene pomnilniške lokacije v drugo poteka tako :

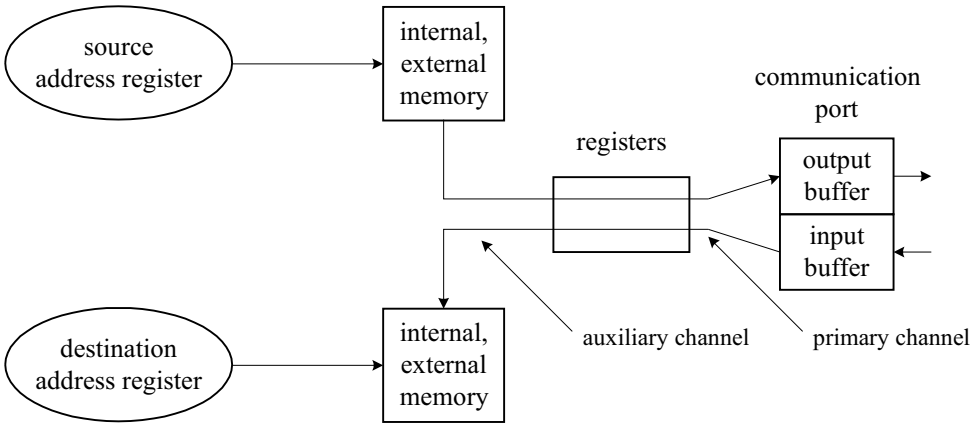
- naslov prve besede niza podatkov se prenese v izvorni naslovni register (source address register).

- naslov končne lokacije prve besede niza podatkov se prenese v ciljni naslovni register (destination addres register).
- velikost niza podatkov je prenešena v števec prenosa (transfer counter).
- indeksni registri (index registers) so naloženi skupaj s povečanjem naslova (adress increment), ki je enako ena, če obstaja zahteva za sekvenčni dostop.
- DMA prenos se sinhronizira s prekinitvami ali preko komunikacijskih vrat. DMA je lahko sinhroniziran glede na izvor, cilj ali glede na oba parametra.
- DMA prebere izvorni podatek in ga prepiše v začasni register (temporary register). Nato se ta vrednost prepiše na ciljni naslov. Indeksni registri se uporabijo za izračun novega naslova izvorne in ciljne lokacije.

Prenos podatkov lahko poteka v primarnem (primary mode) načinu, kar vidimo na sliki 7.10 ali pa v pomožnem (auxiliary mode) načinu, vidno na sliki 7.11.



Slika 7.10: Prenos podatkov v primarnem načinu

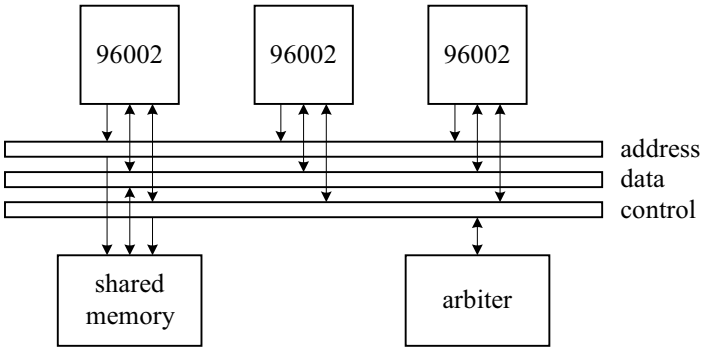


Slika 7.11: Prenos podatkov na pomožnem načinu

Enota za arbitražo vodila (Bus arbiter)

Zunanja (podatkovna in naslovna) vodila se pri DSP-jih večkrat uporabljajo iz vrste razlogov. Preko njih dostopamo do zunanjega pomnilnika (podatkovni ali ukazni), dostopamo lahko tudi do glavnega

(host) računalnika ali pa povezujemo DSP-je v večprocesorske sisteme. Da prihranimo na številu pinov, so vodila največkrat multipleksirana (različni pomnilniki si delijo isti fizični naslov). Vodilo lahko DSP uporablja za dostop do IM/DM drugega DSP-ja preko enote za arbitražo vodila. Uporaba takih vodil omogoča procesorju komunikacijo preko skupnega pomnilnika (shared memory), kar je hitrejše kot preko komunikacijskih vrat. Procesor, ki mu je arbitraža vodila dodelila nadzor nad vodilom je nadrejen (bus master), medtem ko so ostali procesorji podrejeni (bus slave) in čakajo, da jim enota za arbitražo vodila (bus arbiter) dovoli dostop do vodila. Na sliki 7.12 vidimo primer za večprocesorski sistem



Slika 7.12: Večprocesorski sistem

Enota za arbitražo vodila je priključena na sistem preko nadzornega vodila (control bus) in odloča o tem, kateri procesor bo lahko dostopal do vodila če pride do konflikta med le-temi pri hkratni zahtevi po uporabi vodila pa dodeli vodilo po vgrajenem kriteriju.

Nadzorni signali (control signals), ki se uporabljajo so naslednji:

- BREQ - (bus request) zahteva procesorja po vodilu
- BGRANT - enota za arbitražo vodila sporoči procesorju, da je postal nadrejen
- BBUSY - vodilo je zasedeno
- BRECV - procesor sporoči enoti za arbitražo vodila, da je prevzel vlogo nadrejenega
- RECV signal se lahko pojavi le, če je procesor dobil signal BGRANT in pri signalu za prosto vodilo.

Serijske in vzporedne V/I enote

Tipični DSP je povezan z glavnim računalnikom in večkrat prenaša kodo in podatke DSP enoti brez intervencije centralne procesne enote. Za ta namen obstajajo posebne enote t.i. glavni vmesnik, ki ima tako serijska kot vzporedna vrata.

Serijska V/I vrata lahko uporabimo kot vmesnik z glavnim računalnikom ali drugim DSP-jem z nekaj dodatne krmilne logike. Vgrajen je generator ure, ki je sinhroniziran z notranjo uro. DMA kanali so še posebej dobro opremljeni s serijskimi vrati.

Vzporedna vrata imajo paralelni vmesnik za zunanje enote ali druge procesorje in podpirajo dva načina prenosa : DMA in programirani V/I DMA način, ki ne prekinja CPE-ja ko bere ali piše v/iz IM/DM. Paralelna vrata (Parallel Input Output - PIO) imajo tudi asinhron vmesnik za povezavo z zunanjim svetom in sicer s pomočjo uporabe 16-bitnih registrov.

Poglavje 8

Sistolična procesorska polja

8.1 Predstavitev sistoličnih polj

Kot smo spoznali v prejšnjih poglavjih, so sistolična polja vzporedne strukture, sestavljene iz enostavnih procesorskih elementov, povezanih izključno s svojimi najbljižjimi sosedi. Zaradi svoje enostavnosti, modularnosti strukture in enostavnih možnosti za razširitev so tovrstne strukture še posebej primerne za izgradnjo v VLSI tehnologiji. Uporabna so predvsem za reševanje znanstvenih in tehničnih problemov, kjer imamo opravka s posebnimi, predvsem matričnimi algoritmi in kjer zahtevamo visoke hitrosti obdelave podatkov in nizko ceno izvedbe.

Algoritme, ki se na njih izvajajo, imenujemo sistolični algoritmi. Sistolične algoritme uporabljamo na polju sistoličnih celic, na katerih želimo vzporedno izvajati določene operacije. Med izvajanjem sistoličnih algoritmov se zahtevani izračuni odvijajo na posameznih celicah v polju, kot to določajo izbrani algoritmi. Sistolični algoritmi torej določajo

- vrsto operacije, ki naj se v posameznem koraku izvede na posamezni celici,
- obliko povezav med celicami in
- zaporedja vhodnih in izhodnih ukazov ter podatkov na robnih celicah.

Sistolični algoritmi so preračunsko omejeni. Izmenjava podatkov med celicami v polju je lokalna in regularna. Stopnja vzporednosti v izvajanju je torej velika. Vhodno/izhodnih operacij je razmeroma malo, saj se jim zaradi lastnosti sistoličnih polj izogibamo. To omejuje uporabo sistoličnih polj na področje regularnih algoritmov, pri katerih je število potrebnih računskih operacij veliko v primerjavi s potrebami po vhodno/izhodnih povezavah in kjer je stopnja vzporednosti v algoritmih samih visoka. Takšni so predvsem problemi, ki zahtevajo računanje z matrikami.

Običajno sistolična procesorska polja načrtujemo za reševanje točno določenih problemov. To je nov pristop, saj načeloma večprocesorske strukture uporabljamo v splošno uporabnih računalniških sistemih visoke zmogljivosti. Takšni sistemi zahtevajo kompleksen operacijski sistem, ki usklajuje delo posameznih procesorskih enot. Da bi omogočili izvajanje algoritma, potrebujemo veliko število dodatnih operacij, kar zmanjšuje učinkovitost in hitrost delovanja sistema. Programiranje zahteva dobro poznavanje arhitekture in operacijskega sistema na eni in vzporednih algoritmov na drugi strani. Univerzalnost pač nekaj stane.

Zahtevana cena je vendarle previsoka za hitro digitalno obdelavo signalov v realnem času. Visoka pretočnost je pri obdelavi signalov najpomembnejša, zato so za uporabo na tem področju specializirana polja procesorjev najbolj privlačna rešitev. Algoritmom za digitalno obdelavo signalov so običajno skupne lastnosti regularnosti, rekurzivnosti in lokalnih povezav, zato so primerni za realizacijo na sistoličnih procesorskih poljih. Najboljše rezultate dosežemo z realizacijo polj v VLSI tehnologiji. Takšna zgradba je hitra, zmogljiva in relativno cenena. Zmogljivost množice procesorskih enot izkoriščamo

bodisi z dejansko vzporedno obdelavo ali pa s cevovodenjem; najboljši učinek dosežemo s kombinacijo obeh načinov dela. Polje opravlja svojo funkcijo na seriji vhodnih podatkov in tako v liniji za prenos signala nadomesti analogno vezje. Področje uporabe takšnih polj pokriva širok spekter, ki vključuje digitalno filtriranje, spektralno analizo in adaptivno obdelavo signalov, obdelavo slik, seizmičnih, tomografskih in drugih signalov.

Pri načrtovanju sistoličnih procesorskih polj, posebej tistih, izdelanih v VLSI tehnologiji, moramo v čimvečji meri izkoristiti prednosti, ki jih nova tehnologija ponuja. Prav tako moramo računati s težavami zaradi omejenega obsega in hitrosti medsebojnih povezav, zaradi omejenih dimenzij silicijevega čipa ter zaradi omejenega števila priključnih sponk. Oglejmo si nekatera osnovna načela, ki so nam pri načrtovanju v veliko pomoč.

Načelo homogenosti Pri načrtovanju sistoličnih procesorskih polj želimo zgraditi čimbolj regularno in modularno strukturo, s čimer skupno izvedbo poenostavimo. Zaradi regularnosti uporabljenih celic bo cena načrtovanja posameznih procesnih enot in enot pomnilnika bistveno nižja. Do takšne oblike v nekaterih primerih pridemo šele s pozornim študijem algoritma.

Uporaba cevovoda V številnih aplikacijah digitalne obdelave signalov je ključni faktor prepustnost sistema. Če želimo ta parameter izboljšati se cevovodna obdelava naravnost ponuja kot sredstvo za dvig prepustnosti sistema. Metode cevovodne obdelave so že dodobra obdelane, posebej za strojno opremo lahko namreč bistveno izboljšamo lastnosti celotnega sistema.

Načelo lokalnosti Z načelom lokalnosti se srečujemo na vseh stopnjah načrtovanja sistoličnih procesorskih polj v VLSI izvedbi. Izraz pomeni, da se podatki lahko izmenjujejo le med celicami na omejeni medsebojni razdalji. Lokalnost nastopa tako v prostorskem kot tudi v časovnem pogledu. Pojem lokalnosti v načrtovanju procesorskih polj lahko predstavlja lokalnost izmenjave podatkov ali lokalnost pretoka ukazov. Dejansko večina rekurzivnih algoritmov za obdelavo signalov dovoljuje obe vrsti lokalnosti v uporabi predvsem pri načrtovanju polj z valovno fronto.

Ključni faktor - izmenjava podatkov V VLSI tehnologiji si lahko privoščimo veliko preračunavanj. Kritična pa je predvsem izmenjava podatkov. Zato bo v prihodnosti razvoj usmerjen v strukture, ki bodo zmogle z minimalno dodatno logiko uravnotežiti stopnjo preračunske zmogljivosti in stopnjo izmenjave podatkov.

Osnovni problem uporabe sistoličnih procesorskih polj se skriva v načrtovanju ustreznega sistoličnega algoritma, tj. v načinu razdelitve opravil med posamezne celice, ki delo vzporedno opravljajo. Gre za zahtevno opravilo, saj moramo celicam ob ustreznih trenutkih zagotoviti tudi ustrezne vhodne podatke. Predolgo čakanje na slednje namreč izniči vse prednosti, ki smo jih pridobili z uvedbo večjega števila procesnih enot. Posledic, ki jih utegnejo povzročiti neustrezni podatki na vhodih in izhodih celic verjetno ne kaže posebej poudarjati. V praksi se tudi dogaja, da rešujemo probleme na omejenih dimenzijah procesorskih polj, poleg tega pa pogosto dodajamo elemente, s katerimi povečujemo zanesljivost delovanja sistema.

8.2 Model sistoličnega polja

8.2.1 Osnovne definicije

Model procesorskega polja lahko zgradimo za katerokoli množico procesorskih elementov (PE), ki oblikujejo logično celoto, imenovano procesorsko polje (PA). Model ni odvisen od vrste procesorskega sistema.

Pri načrtovanju sistoličnih polj v VLSI tehnologiji, kjer gre za visoko koncentracijo procesorskih elementov na enem samem čipu, se soočamo s problemi, ki za klasične večprocesorske sisteme niso

značilni. Eden od glavnih problemov je število vhodno-izhodnih povezav, ki je omejeno z dimenzijami uporabljenega podnožja integriranega vezja. Drugi glavni problem je dolžina povezav med procesorskimi elementi v polju. Če so povezave med procesorskimi elementi razmeroma dolge glede na njihovo širino, se pojavijo dodatni vplivi njihove upornosti in kapacitivnosti, kar podaljšuje čas za prenos podatkov. Zato morajo biti povezave v VLSI sistoličnih poljih čimkrajše, običajno so med seboj povezani le sosednji procesorski elementi.

Poznamo tudi sistolična polja, v katerih obliko povezav med procesorskimi elementi lahko spremenimo. To nam omogoča posebna kontrolna stikalna mreža (switch lattice), ki vsebuje dodatno nadzorno logiko in ki s pomočjo vgrajenih preklopnih funkcij oblikuje strukturo povezav. Na takšnem vezju moremo zgraditi različne arhitekture (drevo, shuffle mreža, trikotna, kvadratna shema ipd.). Postopek načrtovanja vzporednih algoritmov na takšnih strukturah opravimo v dveh stopnjah. Najprej vzpostavimo komunikacijski graf povezav in šele nato programiramo aktivnosti na posameznih procesorjih.

Za obdelavo signalov pogosto uporabljamo ti. CORDIC procesorske elemente, ki brez dodatnih algoritmov izvajajo osnovne funkcije, potrebne za obdelavo signalov (sin, cos, sqrt, hiperbolične funkcije), poleg tega pa še rotacijo in transformacijo koordinatnih sistemov.

Pri predstavitvi pojmov, povezanih s procesorskimi polji, vzporednimi algoritmi in z ustreznimi transformacijami algoritmov predpostavimo, da:

- procesorski element PE zmore opraviti elementarne funkcije (ki jih zastavlja algoritem) v času enega ciklusa,
- aktiven procesorski element obratuje vsaj eno časovno enoto, zatem pa preda rezultat obdelave sosednjemu procesorju, s katerim ima vzpostavljeno povezavo,
- je položaj vsakega procesorskega elementa v polju določen s kartezičnimi koordinatami.

Vsakemu PE pripada indeksna točka $\mathbf{l}$. Njena vrednost je enaka kartezičnim koordinatam pripadajočega procesorskega elementa. Na tej osnovi oblikujemo indeksno množico procesorskih elementov v procesorskem polju.

Indeksna množica procesorskega polja je $J^M \subset Z^M$, kar predstavlja množico vseh indeksnih točk $\mathbf{l}$ procesorskih elementov v procesorskem polju. Dimenzija točk indeksne množice M lahko zavzame različne vrednosti. Za linearna polja znaša njena vrednost M = 1, za planarna M = 2, itd. Indeksna množica (oz. število indeksnih točk) je omejena, določena pa je s fizičnimi dimenzijami sistoličnega polja.

Pri prenosu algoritma na procesorsko polje moramo vsaki operaciji algoritma določiti procesorski element, na katerem se bo izvajala. Opraviti moramo torej preslikavo iz indeksne množice algoritma v indeksno množico procesorskega polja. Za vsak procesorski element so z algoritmom določeni procesorski elementi, na katere je do naslednjega koraka izvajanja potrebno prenesti rezultate trenutne obdelave. Omenjeni odnosi nam torej določajo komunikacijsko geometrijo polja.

Zaradi lastnosti VLSI sistoličnih polj povezave z bolj oddaljenimi elementi izvedemo preko povezav s sosednjimi elementi, ki se nahajajo na vmesnem prostoru. Zato je osnovni gradnik povezav v sistoličnem polju vez s sosednjim elementom. Takšno povezavo imenujemo povezovalni koren.

Povezovalni koren je vektor, ki predstavlja razliko indeksnih točk sosednjih procesorskih elementov v sistoličnem polju, med katerima je vzpostavljena poveza. Naj bosta

$$\mathbf{l}_v,\ \mathbf{l}_w \in J^M$$

oznaki med seboj povezanih sosednjih procesorskih elementov v polju J^M. Če povezava poteka v smeri $\mathbf{l}_v$ proti $\mathbf{l}_w$ je povezovalni koren stolpni vektor

$$\mathbf{l}^{(p)} = \mathbf{l}_w - \mathbf{l}_v$$

Povezovalni koreni oblikujejo povezovalno matriko $\mathbf{P} \in \mathcal{Z}^{(n-1)\times r}$. Dimenzije povezovalnih primitivov določajo dimenzije indeksne množice procesorskega polja.

Povezovalna matrika $\mathbf{P}$ procesorskega polja je matrika

$$\mathbf{P} = (\mathbf{l}_1^{(p)}, \ldots, \mathbf{l}_M^{(p)})_r^{\mathrm{T}} \subset \mathcal{Z}^{M\times r} = \{\mathbf{l}_1^{(p)}, \ldots, \mathbf{l}_r^{(p)}\}$$

pri čemer imamo v polju r povezovalnih korenov.

Povezovalni koreni so definirani z množico r različnih stolpnih vektorjev, ki določajo smeri povezav med procesorskimi elementi. Dimenzija vektorjev je enaka M. Povezovalno matriko $\mathbf{P}$ sestavljajo celoštevilske vrednosti, določene s smermi povezav med procesorskimi elementi. S pomočjo matrike $\mathbf{P}$ lahko opišemo tudi bolj zapletene povezave med procesorskimi elementi, ki niso sosednji.

Primer 1 *Na planarnem dvodimenzionalnem kvadratnem polju povežimo vsak element s štirimi sosednjimi procesorskimi elementi (z levim, z desnim, z zgornjim in s spodnjim). Ob predpostavki, da prva koordinata določa pomik v levo ali v desno glede na obravnavani procesor, sta ustrezna povezovalna korena enaka* $\mathbf{l}_1^{(p)} = (1,0)^{\mathrm{T}}$ *in* $\mathbf{l}_2^{(p)} = (-1,0)^{\mathrm{T}}$. *Povezavo z zgornjim in spodnjim procesorjem nam določata korena* $\mathbf{l}_3^{(p)} = (0,1)^{\mathrm{T}}$ *in* $\mathbf{l}_4^{(p)} = (0,-1)^{\mathrm{T}}$. *Povezavo s samim seboj nakažimo s korenom* $\mathbf{l}_5^{(p)} = (0,0)^{\mathrm{T}}$. *Na osnovi navedenih povezovalnih korenov oblikujemo povezovalno matriko procesorskega polja:*

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}.$$

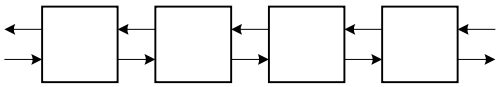
Procesorsko polje je torej popolnoma določeno s parom

$$(J^M, \mathbf{P}).$$

8.2.2 Nekaj primerov procesorskih polj

Oglejmo si nekaj primerov povezav v procesorskih poljih.

Linearno polje



Slika 8.1: Linearno procesorsko polje

Slika 8.1 predstavlja linearno polje, v katerem procesorski elementi izmenjujejo podatke s svojimi levimi in desnimi sosedi. Mejni elementi na koncu polja imajo le po enega soseda, ki se nahaja v notranjosti polja. Pretok podatkov lahko poteka v eni ali v obeh smereh. Povezovalna matrika za dvosmerni prenos podatkov ima obliko

$$\mathbf{P} = \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

Kadar procesor operira tudi s podatki, ki so v njem stacionirani, pravimo, da je povezan sam s seboj. V takšnem primeru je povezovalna matrika enaka

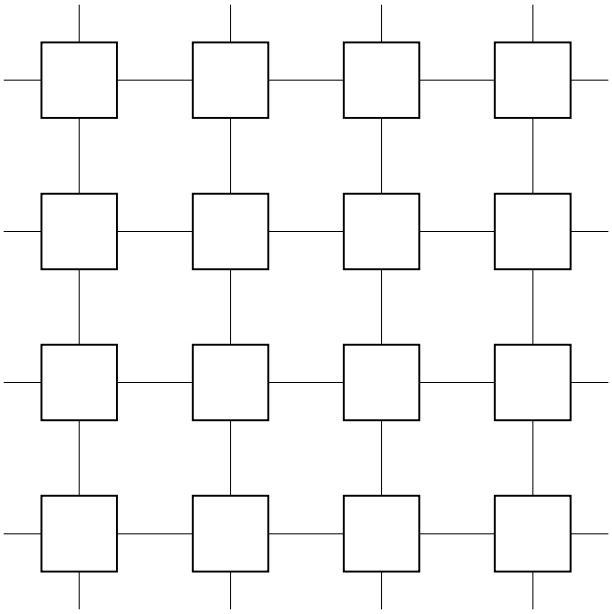
$$\mathbf{P} = \begin{bmatrix} 0 & 1 & -1 \end{bmatrix}.$$

Kadar pa je tok podatkov enosmeren, imamo

$$\mathbf{P} = \begin{bmatrix} 0 & 1 \end{bmatrix}.$$

Linearna polja so primerna za operacije konvolucije, DFT, karterzičnega produkta, za reševanje enačb oblike $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ ipd.

2D kvadratno polje



Slika 8.2: 2D kvadratno polje

Procesorski elementi so povezani v kvadratno polje (slika 8.2) tako, da ima vsak notranji element polja 4 sosede. Elementi na stranicah imajo po 3, vogalni pa po dva soseda. Dovoljen je pretok podatkov v vseh 4 smereh, tako da dobimo naslednjo povezovalno matriko:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \end{bmatrix}.$$

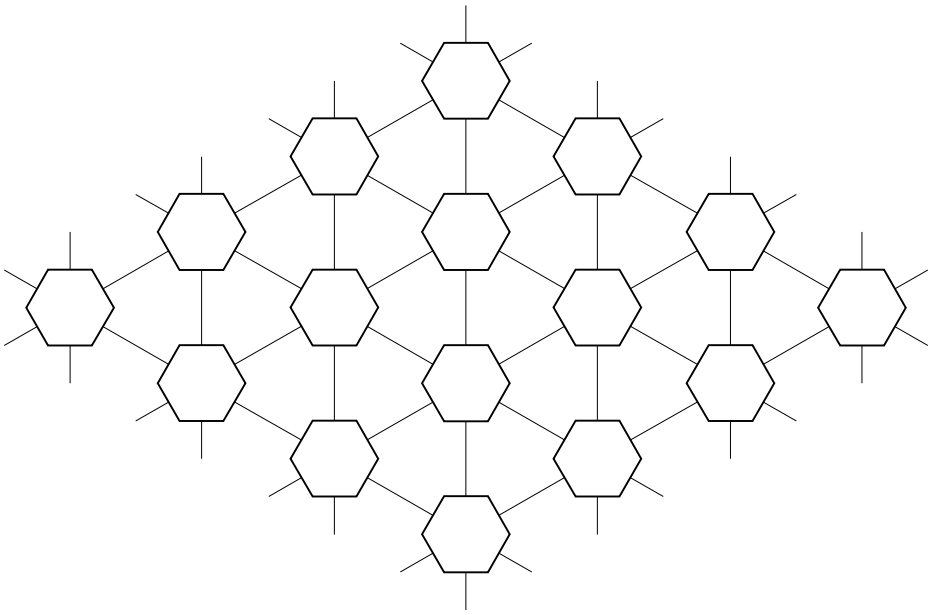
V matriki je opredeljena tudi povezava elementa s samim seboj. V primeru enosmernega toka podatkov ni povezav z negativnimi elementi, kot tudi ne povezav elementov s samim seboj. Povezovalna matrika je v tem primeru enaka

$$\mathbf{P} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

2D kvadratna polja so primerna za množenje, QR in LU dekompozicijo matrik, za dinamično programiranje, za računanje konvolucije, DFT, za reševanje enačb oblike $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$ itd.

2D heksagonalno polje

Razporeditev elementov polja in medsebojnih povezav spominja na satovje v panju, kar si lahko ogledamo na sliki 8.3. Vsak notranji PE ima po 6 sosedov, PE na stranicah po 4, vogalni pa po 2



Slika 8.3: 2D heksagonalno polje

soseda. Teoretično bi lahko uporabili dvosmerni pretok podatkov v vseh šestih smereh, vendar bi bila takšna zgradba zapletena, zato najpogosteje uporabljamo pretok podatkov v treh smereh in kvečjemu še povezavo s stacionarnimi podatki v samem PE. Takemu primeru ustreza naslednja matrika:

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix}.$$

Enosmerni pretok podatkov ne dovoljuje negativnih elementov povezovalnih korenov in ne povezave PE s samim seboj:

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

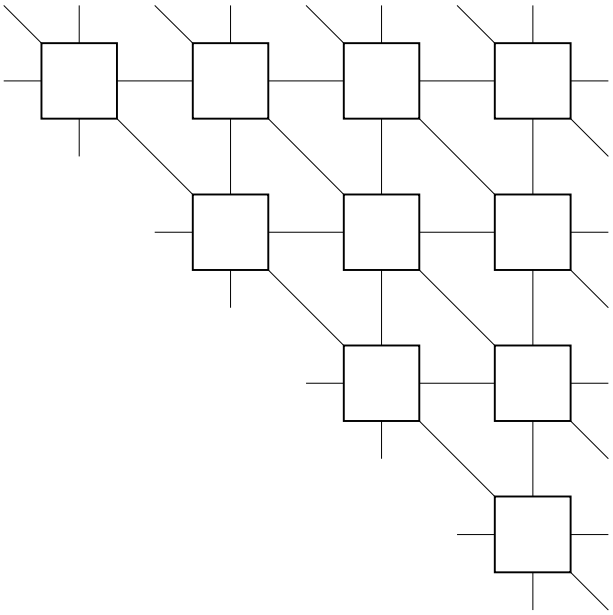
2D heksagonalna polja uporabljamo za množenje matrik, QR in LU dekompozicijo in drugo matrično aritmetiko, za izvedbo konvolucije, DFT ter pri operatorjih za relacije baze podatkov.

2D trikotno polje

2D trikotno polje (na sliki 8.4) je po obliki enako polovici 2D kvadratnega polja, deformiranega v romboid. Vsak notranji PE ima po 6, stranični po 4 in vogalni po 2 soseda. Kot pri heksagonalnem polju je tudi v tem primeru dovoljen dvosmeren pretok podatkov v šestih smereh, vendar se tako zapleteni obliki izogibamo. Najpogosteje uporabljamo povezave v treh smereh in kvečjemu še povezave s samim seboj, tako da je povezovalna množica 2D trikotnega polja enaka

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

Trikotno polje je samo po sebi enosmerno, kadar elementi niso povezani s samim seboj. Povezava po diagonali ni nujno potrebna, v tem primeru imamo



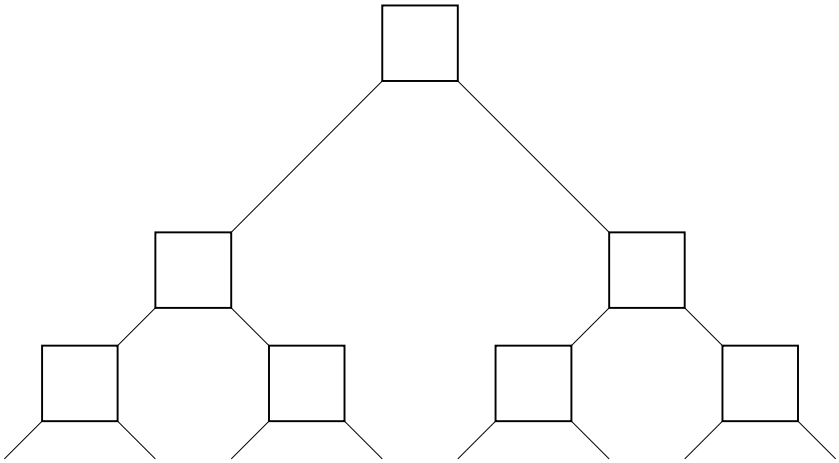
Slika 8.4: 2D trikotno polje

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}.$$

Trikotno polje uporabljamo za invertiranje trikotnih matrik in splošno v matrični aritmetiki, za prepoznavanje formalnih jezikov, za dinamično programiranje in drugje.

Omenimo še, da je z uporabo posebnih transformacij mogoče pretvoriti algoritem za kvadratno, trikotno ali heksagonalno polje v katerikoli od teh treh oblik.

Drevo



Slika 8.5: Drevo

Procesorski elementi z medsebojnimi povezavami tvorijo strukturo drevesa (slika 8.5). Vsaki notranji element ima po 2 soseda, končni elementi pa sosedov nimajo. To so običajno vhodni ali izhodni

PE. Tok podatkov je lahko dvosmeren ali enosmeren, v smeri od zgoraj navzdol ali od spodaj navzgor. Matrika za povezovanje podatkov ima obliko

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Povezava elementov s samim seboj ni nujna, v tem primeru je povezovalna matrika enaka

$$\mathbf{P} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Polja v obliki drevesa so namenjena za reševanje rekurzivnih problemov, za sortiranje itd.

Spoznali smo le nekaj načinov za povezovanje elementov v procesorskih poljih. Na voljo je seveda še več možnosti povezovanja, tudi v več dimenzijah. Omejili smo se na nekaj planarnih in linearnih struktur, ki so tudi najbolj primerne za izvedbo sistoličnih polj, predvsem v VLSI tehnologiji obdelave.

8.2.3 Zapis aktivnosti procesorskih elementov v procesorskih poljih

Oglejmo si še zapis, s pomočjo katerega spremljamo aktivnosti procesorskih elementov v procesorskem polju. V ta namen uporabimo matriko $\mathbf{M}$, v kateri z oznako 0 označimo neaktiven, z oznako 1 pa aktiven procesorski element.

Da bi lažje spremljali aktivnosti na elementih procesorskega polja, uporabimo matriko $\mathbf{M}(t) = \{m_{ij}(t)\}_{n\times m}$. Poleg nje potrebujemo še matriko $\mathbf{Q}(t)_{n\times m}$, katere elementi so nenegativna cela števila, označujejo pa zaporedno število operacij, ki se izvaja na posameznem procesorskem elementu. Dimenzije obeh matrik so seveda enake dimenzijam procesorskega polja.

Začetna vrednost matrik znaša

$$\mathbf{M}(0) = \mathbf{Q}(0) = 0_{n\times m}$$

kar pomeni, da v času $t = 0$ niti en procesorski element ni aktiven in da v polju ni bila izvršena nobena operacija.

Vrednosti obeh matrik so v nadaljevanju odvisne od uporabljenega algoritma in od pretočnosti operacij. Običajno se v času $t = 1$ aktivira procesor s koordinatami $(0, 0)^{\mathrm{T}}$, naslednje operacije pa potekajo tako, da aktivni procesorski elementi vzbujajo svoje sosede ter jim zagotavljajo podatke za regularno izvajanje aktivnosti.

$$\mathbf{M}(1) = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} = \mathbf{Q}(1)$$

Na ta način se oblikuje val, ki se širi vzdolž procesorskega polja.

Seveda je povsem mogoče, da se v času $t = 1$ aktivira več procesorskih elementov, na primer vsi elementi prve vrstice. V vsakem primeru ima matrika $\mathbf{M}(1)$ vrednosti 1 povsod, kjer so procesorski elementi v času $t = 1$ aktivni.

$$\mathbf{M}(1) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} = \mathbf{Q}(1)$$

Trenutne aktivnosti procesorskih elementov so lahko glede na uporabljeni algoritem k-stopenjsko rekurzivno odvisne od predhodnih aktivnosti:

$$\mathbf{M}(t) = f(\mathbf{M}(t-1), \ldots, \mathbf{M}(t-k))$$

Matrika $\mathbf{Q}(t)$ je po definiciji kumulativna aktivnost, določena z relacijo

$$\mathbf{Q}(t) = \mathbf{Q}(t-1) + \mathbf{M}(t).$$

Elementi $q_{ij}(t)$ matrike $\mathbf{Q}(t)$ predstavljajo število operacij, ki jih je določeni procesorski element opravil do trenutka t. Očitno je, da je vrednost elementa $q_{ij}(t)$ kljub $m_{ij}(t) = 0$ lahko od 0 različna. To le pomeni, da ustrezni procesorski element v trenutku t ni aktiven in da po pripadajočem algoritmu čaka na podatke s sosednjega procesorskega elementa.

Uvedba matrik $\mathbf{M}(t)$ in $\mathbf{Q}(t)$ nam olajša analizo algoritmov in časovnih aktivnosti na procesorskih elementih. Omogoča nam vpogled v trenutno delovanje procesorskih elementov v polju, poenostavi časovno analizo izvajanja vzporednih algoritmov v procesorskih poljih ter olajša načrtovanje algoritmov s posebnimi zahtevami (avtomatsko opravljanje napak).

V nadaljevanju si oglejmo značinosti algoritmov, ki so bistvenega pomena za preoblikovanje algoritma za vzporedna procesorska polja.

8.3 Preslikava algoritmov na procesorske sisteme

V tem in naslednjem podpoglavju se bomo posvetili algoritmom in možnostim transformacije algoritmov v obliko, ki omogoča izvajanje na paralelnem procesorskem sistemu oziroma procesorskem polju.

Prvi korak pri transformaciji algoritma je vzporedna izražava algoritma. Do tako izraženega algoritma lahko pridemo na dva načina:

- z vektorizacijo sekvenčno izraženega algoritma ali
- z neposredno izražavo algoritma v vzporedni obliki.

8.3.1 Vektorizacija sekvenčno izraženega algoritma

Običajni računalniki vsebujejo le eno CPE, ki algoritme izvaja sekvenčno. Večina programov je napisana v višjih programskih jezikih in jih zato moremo s posebnim vektorskim prevajalnikom prevesti tako, da v njih poiščemo dele, ki bi se lahko izvajali sočasno. Takšen prevajalnik oblikuje sekvenčno kodo in, kjer je to mogoče, še dodatno paralelno kodo. Če uporabljamo programski jezik FORTRAN, se prevajalnik pri iskanju paralelizma osredotoči na DO zanke.

8.3.2 Algoritmi, izraženi neposredno v paralelni obliki z enkratno prirejenimi spremenljivkami

Izdelava vektorskega prevajalnika je zelo zahtevna in zapletena naloga, pa tudi njegovo delovanje pogosto ni najbolj uspešno. Zato mora uporabnik že kar med pisanjem programa uporabiti ukaze za opis paralelnosti algoritma. Prvi pogoj, ki ga moramo izpolniti, če želimo določeni algoritem izvajati na paralelnem procesorskem sistem, je enkratna prireditev spremnljivk (single assignment). Ker je v spodnjem FORTRAN-skem programu C(I) prirejen večkrat,

```
DO 10 I=1,4
C(I)=0
DO 10 J=1,4
C(I)=C(I)+A(I,J)*B(J)
10   CONTINUE
```

mu dodamo še dodatni indeks J in tako dosežemo enkratno prireditev C(I,J).

```
DO 10 I=1,4
C(I,1)=0
DO 10 J=1,4
C(I,J+1)=C(I,J)+A(I,J)*B(J)
10   CONTINUE
```

Pogosto se srečujemo z algoritmi v obliki rekurzivnih enačb, za katere je značilno, da imajo vse spremenljivke enkratno prirejene. Oglejmo si ponovno primer množenja matrike z vektorjem $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$:

$$c_i^{(j+1)} = c_i^{(j)} + a_i^{(j)} b_i^{(j)}$$

j je indeks rekurzije in

$$c_i^{(1)} = 0$$

$$a_i^{(j)} = A(i, j)$$

$$b_i^{(j)} = B(j).$$

Rekurzivna enačba uporablja časovne in prostorske indekse. Indeks j lahko uporabimo kot časovni in indeks i kot prostorski, vlogi indeksov pa lahko tudi zamenjamo.

8.3.3 Postopki transformiranja algoritmov

Postopek transformiranja algoritma z enkratno prirejenimi spremeljivkami obsega naslednje tri korake:

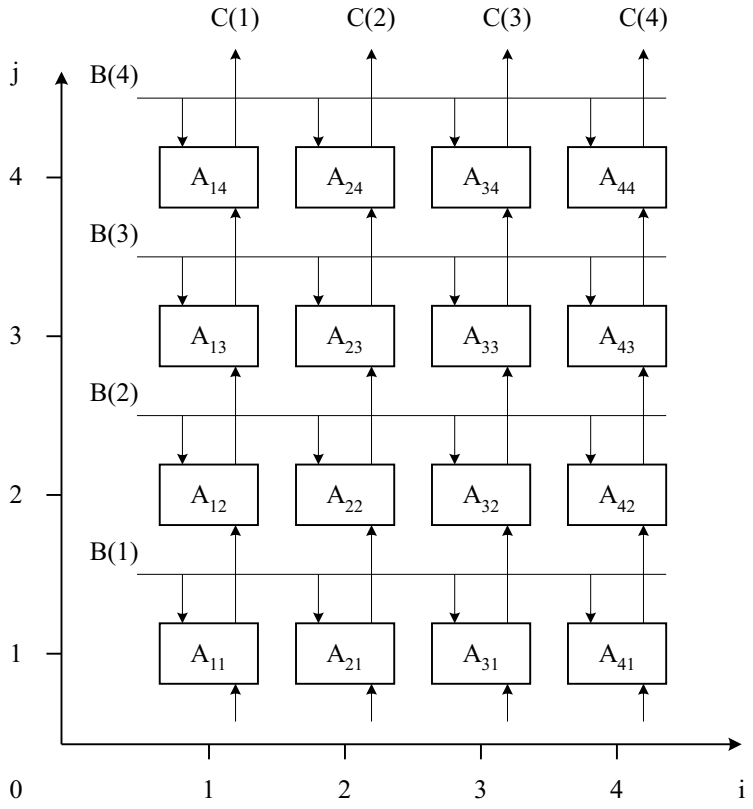
- oblikovanje grafa odvisnosti,
- oblikovanje grafa poteka signalov,
- oblikovanje primernega vzporednega procesorskega sistema.

Graf odvisnosti (dependance graph)

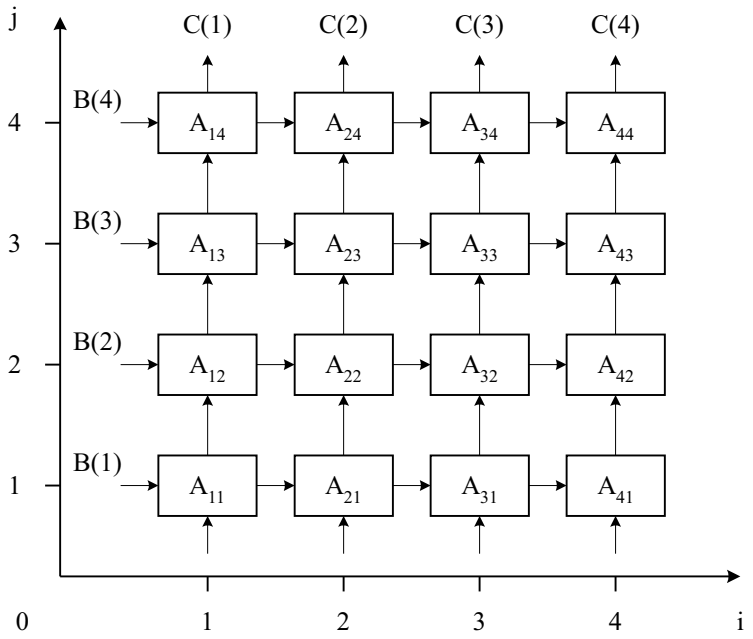
Če želimo popolnoma izkoristiti paralelizem v uporabljenih algoritmih, moramo temeljito preučiti odvisnost posameznih spremenljivk v vseh operacijah. Kadar operacije v sekvenčnem algoritmu uporabljajo spremeljivke, ki so neodvisne, jih lahko izvajamo paralelno.

Osnovna določila grafa Graf G = (N, A) je sestavljen iz množice vozlišč N in iz množice povezav A. Vsaka povezava veže dve vozlišči, vendar samo v eno smer. Pot je zaporedje povezav.

Graf odvisnosti Graf odvisnosti je torej graf, ki prikazuje odvisnost izračunov, oziroma operacij znotraj nekega algoritma. To je grafična ponazoritev izvajanja algoritma, ki ima enkratno prirejene spremenljivke. Vsako vozlišče z določenimi indeksi predstavlja določeno operacijo algoritma, v kateri se uporabljajo spremenljivke z istimi indeksi. Do grafa najlaže pridemo, če uporabimo rekurzivne enačbe. Graf odvisnosti postopka množenja matrike z vektorjem je prikazan na sliki 8.6.



Slika 8.6: Graf odvisnosti pri množenju matrike z vektorjem: globalne povezave



Slika 8.7: Graf odvisnosti pri množenju matrike z vektorjem: lokalne povezave

Lokaliziran graf odvisnosti Na sliki 8.6 je razvidno, da je spremenljivka $B(j)$ hkrati oddana vozliščem z enakim indeksom j. Takšnim spremenljivkam pravimo oddane spremenljivke in takšnim

povezavam globalne povezave. Dostikrat lahko takšne povezave zamenjamo z enostavnejšimi lokalnimi povezavami. Pravimo, da je algoritem lokaliziran, kadar so vse spremenljivke v posameznem vozlišču odvisne le od spremeljivk v sosednjih vozliščih. Na sliki 8.7 se $B(j)$ razširja po korakih do vseh vozlišč, ki imajo enak indeks j.

Prilagojen lokaliziran algoritem bi bil:

```
 DO 10 I=1,4
 C(I)=0
 DO 10 J=1,4
 B(I+1,J)=B(I,J)
 C(I,J+1)=C(I,J)+A(I,J)*B(I+1,J)
10   CONTINUE
```

Graf poteka signalov (signal flow graph)

Do grafa poteka signalov pridemo tako, da v določeni smeri preslikamo graf odvisnosti. Glede na različne možnosti preslikave grafa odvisnosti lahko iz enega grafa odvisnosti dobimo več različnih grafov poteka signalov. V dobljenem grafu poteka signalov prav tako nastopajo vozlišča in povezave (pa tudi zakasnitve), vendar so vozlišča v tem grafu že kar bodoče procesorske enote. Na procesorske enote bi lahko preslikali vozlišča grafa odvisnosti, vendar bi nas to pripeljalo do zelo slabe izkoriščenosti procesorskih enot, ker bi bile le-te obremenjene le določen čas. Zato običajno več vozlišč grafa odvisnosti preslikamo v eno vozlišče grafa poteka signalov. Vozlišča grafa poteka signalov so označena s pravokotniki in predstavljajo izvajanje aritmetičnih in logičnih operacij, ki naj se izvedejo neskončno hitro. Povezave ponazarjajo odvisnosti med vozlišči in so brez zakasnitve ali z zakasnitvijo D. V primerjavi z grafom odvisnosti ima graf poteka signalov naslednje lastnosti:

- je bolj zgoščen kot graf odvisnosti,
- je bliže strojni realizaciji,
- povezave lahko imajo zaksnitve.

Razdelitev in vrstni red operacij Pri preslikavi iz grafa odvisnosti v graf poteka signalov moramo biti pozorni predvsem na dvoje:

- katere operacije oziroma vozlišča grafa odvisnosti priredimo določeni procesorski enoti oziroma vozlišču v grafu poteka signalov
- v kakšnem vrstnem redu procesorska enota oziroma vozlišče v grafu poteka signalov izvede njej prirejene operacije.

a) Prirejanje operacij vozliščem v grafu poteka signalov (processor assignment) Graf odvisnosti preslikamo v smeri vektorja **d** (smer poljubno izmeremo) v graf poteka signalov (slika 8.8). Z uporabo tega postopka zmajšamo dimenzijo grafa za ena, kot je razvidno s slike, kjer smo dvodimenzionalni prostor grafa odvinosti preslikali v enodimenzionalni prostor grafa poteka signalov. Pri tem smo dodali zakasnitev v lastne zanke vozlišč. S tem določene operacije algoritma preslikamo na vozlišče grafa poteka signalov.

b) Določanje vrstnega reda izvajanja operacij v vozlišču grafa poteka (scheduling) Na sliki 8.9 je predstavljen možni vrstni red izvajanja operacij v grafu odvisnosti, kjer so vsa vozlišča, ki so aktivna istočasno, povezana s hiperravninami. Vrstni red izvajanja ponazorimo z vektorjem **s**, ki je pravokoten na hiperravnine in ne sme biti pravokoten na vektor projekcije **d**. Lahko bi rekli, da

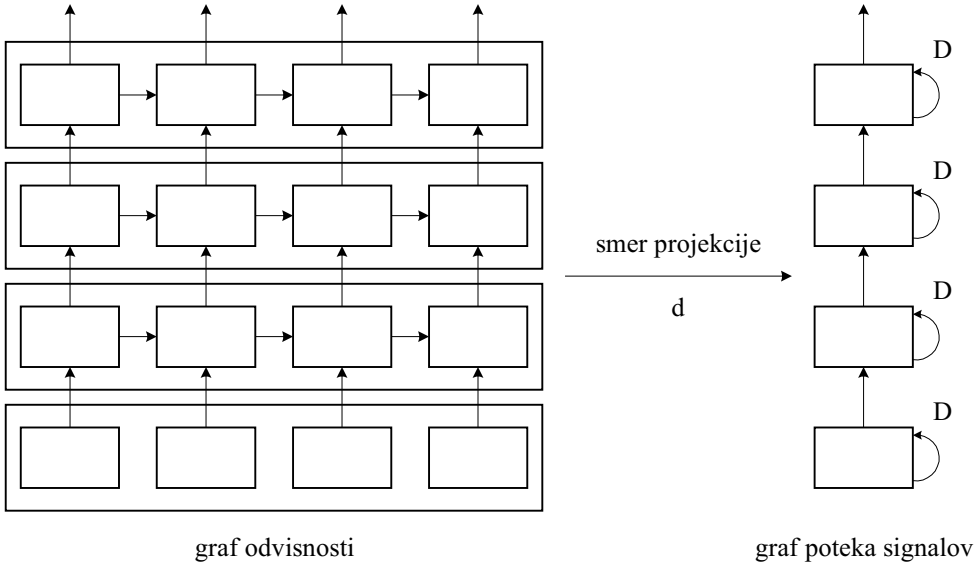
je določanje vrstnega reda operacij v vozlišču graf poteka signalov prav tako odvisno od preslikave. Zadostiti moramo pogojema

$$\mathbf{s}^{\mathrm{T}} \cdot \mathbf{e} \geq 0 \quad \text{in} \quad \mathbf{s}^{\mathrm{T}} \cdot \mathbf{d} > 0, \tag{8.1}$$

kjer je $\mathbf{e}$ vektor poljubne povezave v grafu odvisnosti. Glede na graf odvisnosti in vektor projekcije $\mathbf{d}$ ločimo več načinov določanja vrstnega reda operacij:

- normalni vrstni red, kadar je vektor $\mathbf{s}$ vzporeden z vektorjem $\mathbf{d}$,
- rekurzivni vrstni red, kadar je vektor $\mathbf{s}$ vzporeden z eno od indeksnih osi grafa odvisnosti,
- sistolični vrstni red, kadar je v vsaki povezavi grafa poteka signalov vsaj ena zakasnitev.

Spomnimo se primera množenja matrike in vektorja $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$ s slike 8.7. Če preslikamo ta dvodimenzionalni graf odvisnosti v smeri (0,1), z uporabo normalnega vrstnega reda operacij, pridemo do enodimenzionalnega grafa poteka signalov, ki ga vidimo na sliki 8.10. Na sliki 8.12 je prikazan graf poteka signalov pri projekciji v smeri (1,0) in na sliki 8.12 isti graf v sistolični izvedbi. V tem primeru podatki vstopajo zakasnjeno, kar je posledica zakasnitve med vozlišči.



Slika 8.8: Linearna projekcija v smeri vektorja $\mathbf{d}$

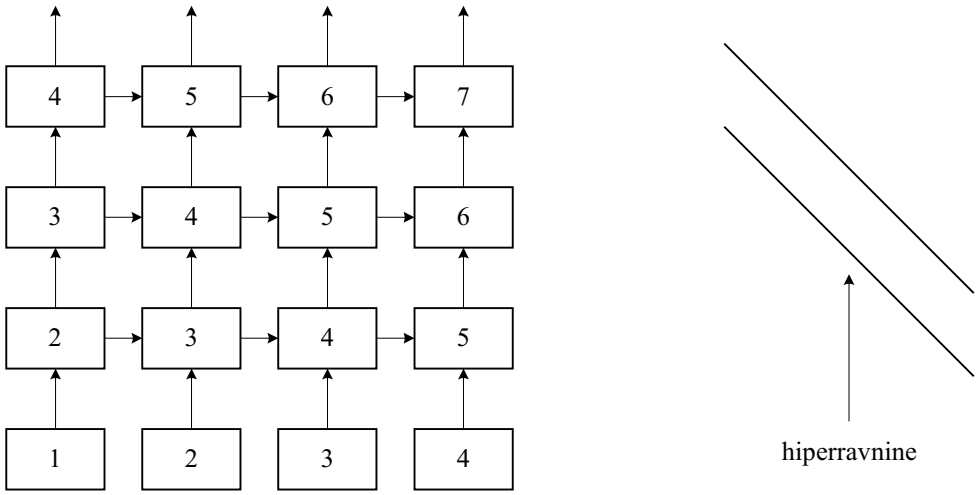
Definicija preslikav Vzemimo n-dimenzionalni graf odvisnosti, izbrani vektor projekcije $\mathbf{d}$ in vektor $\mathbf{s}$, ki mora biti v skladu z neenačbama 8.1. Graf poteka signalov nato preoblikujemo s pomočjo preslikav vozlišč in preslikave povezav.

a) Preslikava vozlišč Graf poteka signalov je dimenzije $(n-1)$, pri čemer vektor $\mathbf{c}$, reda $n \times 1$, vsebuje koordinate, ki nastopajo v grafu odvisnosti, in vektor $\mathbf{n}$, reda $(n-1) \times 1$, koordinate, ki po preslikavi nastopajo v grafu poteka signalov. Povezuje ju enačba

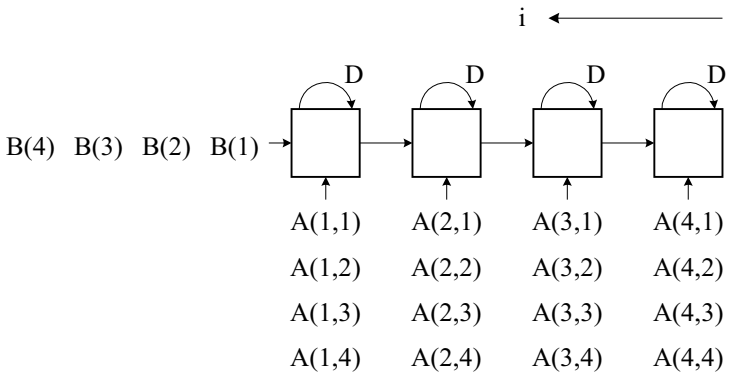
$$\mathbf{n} = \mathbf{P}^{\mathrm{T}} \cdot \mathbf{c},$$

kjer je $\mathbf{P}$ matrika baze, reda $(n-1) \times n$, za katero velja

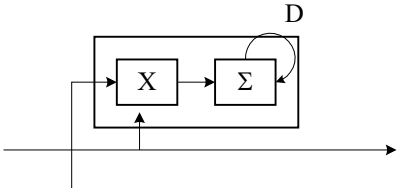
$$\mathbf{P}^{\mathrm{T}} \cdot \mathbf{d} = 0.$$



Slika 8.9: Vrstni red izvajanja operacij, prikazan v obliki hiperravnin



Slika 8.10: Graf poteka signalov $\mathbf{d}^{\mathrm{T}} = (0, 1)$



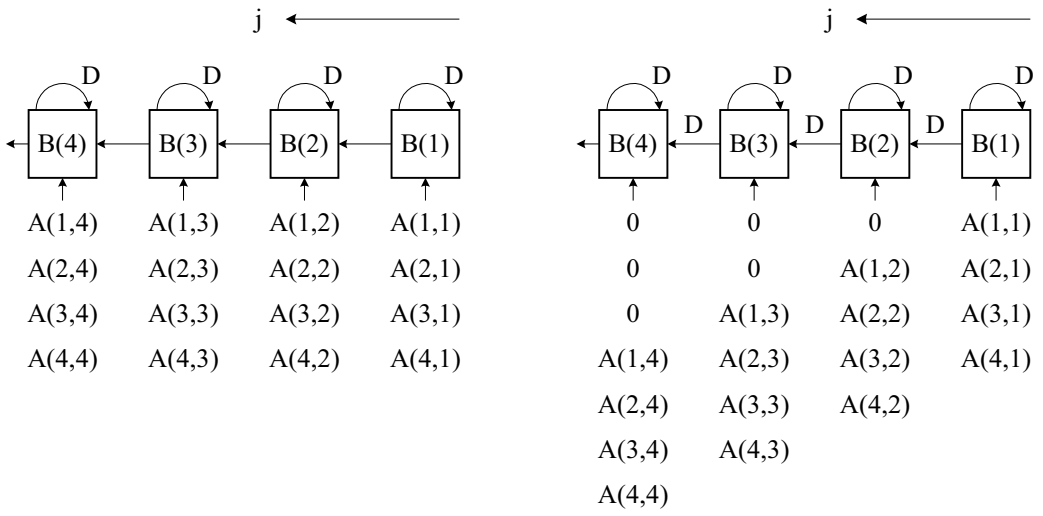
Slika 8.11: Operacija znotraj vozlišča

b) Preslikava povezav Ta preslika povezave v grafu odvisnosti v povezave v grafu poteka signalov.

$$\left[\frac{\mathrm{D}(\mathbf{f})}{\mathbf{f}}\right] = \left[\frac{\mathbf{s}^{\mathrm{T}}}{\mathbf{P}^{\mathrm{T}}}\right] \cdot \mathbf{e} \tag{8.2}$$

e je vektor povezave v grafu odvisnosti, ki se v grafu poteka signalov preslika v povezavo, katero ponazarja vektor **f**. D(**f**) je celo število, ki določa zakasnitev povezave **f**. Vektor **e** je reda $n \times 1$ in vektor **f** $(n-1) \times 1$.

c) Primer preslikave Vzemimo spet primer množenja matrike z vektorjem $\mathbf{c} = \mathbf{A} \cdot \mathbf{b}$. Graf odvisnosti je prikazan na sliki 8.7. Ta graf je dvodimenzionalen s koordinatama oziroma indeksoma i



Slika 8.12: levo: graf poteka signalov $\mathbf{d}^{\mathrm{T}} = (0, 1)$, desno: isti graf v sistolični obliki

in j. Zato ima vektor $\mathbf{c}$ obliko $\mathbf{c}^{\mathrm{T}} = (i, j)$. Naj velja, da je

$$\mathbf{d} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Glede na enačbo $\mathbf{n} = \mathbf{P}^{\mathrm{T}} \cdot \mathbf{c}$ velja

$$j = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

in glede na enačbo $\mathbf{P}^{\mathrm{T}} \cdot \mathbf{d} = 0$

$$\begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0.$$

Naj bo vektor $\mathbf{s}^{\mathrm{T}} = (1, 1)$, (ta je pravokoten na hiperravnine grafa odvisnosti, torej so razmere popolnoma enake kot na sliki 8.8) in naj izpolnjuje neenačbi $\mathbf{s}^{\mathrm{T}} \cdot \mathbf{e} \geq 0$ in $\mathbf{s}^{\mathrm{T}} \cdot \mathbf{d} > 0$, pri čemer je v grafu odvisnosti $\mathbf{e}_1^{\mathrm{T}} = (1, 0)$ in $\mathbf{e}_2^{\mathrm{T}} = (0, 1)$. Glede na enačbo 8.2 definirajmo še preslikavo povezav:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Zaključimo lahko, da smo graf odvisnosti preslikali na premico j in dobili dve povezavi : $f_1 = (0)$ in $f_2 = (1)$ z zakasnitvama $\mathrm{D}(f_1) = 1$ in $\mathrm{D}(f_2) = 1$. Povezava f_1 predstavlja kar lastno zanko vozlišča z enotino zakasnitvijo D. Ustrezen graf poteka signalov je prikazan na sliki 8.12.

8.4 Tehnike načrtovanja sistoličnega procesorskega polja

Sistolično procesorsko polje je sestavljeno iz procesorskih elementov, ki ritmično izvajajo algoritem in ritmično prenašajo podatke. Njegove najpomembnejše lastnosti so: modularnost, regularnost, lokalne povezave, vzporednost, cevovodni sistem in visoka stopnja sinhronizacije. Hitrost sistoličnega polja (velja tudi za ostale procesorske sisteme) izrazimo kot kvocient hitrosti algoritma na običajnem sekvenčnem računalniku in hitrosti na sistoličnem procesorskem polju.

8.4.1 Sinhronizacija polja

Celotno polje nadzira ena sama ura, ki mora opravljati dve nalogi:

- kot običajna ura določa takt delovanja procesorskih elementov in
- določa takt delovnja polja, to je trenutke, ob katerih se izvrši prenos podatkov med procesorskimi elementi.

Zaradi njene pomembnosti lahko rečemo, da je najšibkejši člen v sistoličnem procesorskem polju. Posebno pri velikih poljih se lahko pojavi zakasnitev urinega signala, če so fizične povezave do posameznih procesorskih elementov različno dolge. Temu se izognemo z enako dolgimi povezavami ali uporabimo drugače sinhronizirano polje - "wavefront" procesorsko polje.

8.4.2 Preslikava grafa odvisnosti in grafa poteka signalov na sistolično procesorsko polje

V poglavju 8.3 smo spoznali, kako pridemo do grafa poteka signalov. V tem poglavju pa smo podali prehod iz grafa poteka signalov na sistolično procesorsko polje. Možna je tudi neposredna preslikava iz grafa odvisnosti na sistolično procesorsko polje.

Neposredna preslikava grafa odvisnosti na sistolično procesorsko polje

Preslikava graf odvisnosti na sistolično procesorsko polje je v bistvu le posebna oblika preslikave grafa odvisnosti na graf poteka signalov (vsaka povezava v grafu poteka signalov mora imeti vsaj eno zakasnitev). Pri običajni preslikavi smo morali zadostiti pogojema $\mathbf{s}^{\mathrm{T}} \cdot \mathbf{e} \geq 0$ in $\mathbf{s}^{\mathrm{T}} \cdot \mathbf{d} > 0$. $\mathbf{s}$ je vektor, ki je pravokoten na hiperravnine oziroma ravnine istočasnih operacij v grafu odvisnosti, $\mathbf{d}$ je vektor, ki kaže smer projekcije in $\mathbf{e}$ je vektor poljubne povezave med vozlišči v grafu odvisnosti. Ti dve neenačbi zagotavljata, da bo zakasnitev v povezavah grafa poteka signalov enaka ali večja od nič. Ker pa mora imeti sistolično procesorsko polje zakasnitve večje od nič, spremenimo gornji dve neenačbi v

$$\mathbf{s}^{\mathrm{T}} \cdot \mathbf{e} \geq 0 \quad \text{in} \quad \mathbf{s}^{\mathrm{T}} \cdot \mathbf{d} > 0$$

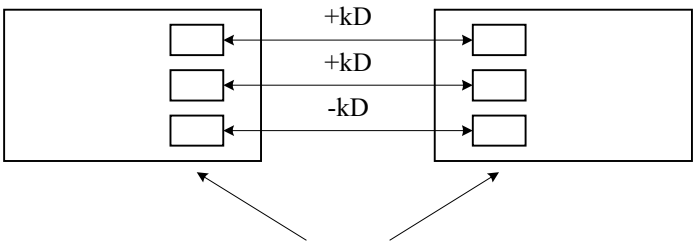
Preslikava grafa poteka signalov na sistolično procesorsko polje

Do sistoličnega procesorskega polja pridemo tako, da v poljubni graf poteka signalov uvedemo cevovodni sistem (pipelining), oziroma da v njem preprečimo povezave brez zakasnitve. V pomoč sta nam postopek časovnega skaliranja in postopek presečnih množic.

Postopek časovnega skaliranja Zakasnitve povezav v grafu poteka signalov lahko pomnožimo z določeno konstanto α (cevovodna perioda), tako da zakasnitve D preidejo v αD. α mora biti naravno število. Glede na konstanto α moramo dodati premore v vhodno/izhodnih nizih podatkov, ki vstopajo ali izstopajo iz grafa poteka signalov.

Postopek presečnih množic in prenos zakasnitev Presečna množica v grafu poteka signalov je množica povezav, ki razdeli graf na dva dela. Po oblikovanju presečne množice lahko povezave razdelimo na tiste, ki gredo levo in tiste, ki gredo desno, kot prikazuje slika 8.13. Po pravilu prenosa zakasnitev lahko povezavam, ki gredo v določeno smer, prištejemo k zakasnitev in jih odštejemo povezavam, ki gredo v nasprotno smer, ali obratno. Vhodno/izhodni nizi podatkov se ne spremenijo, če se nahajajo na isti strani presečne množice, v nasprotnem primeru pa moramo tudi njih zakasniti oziroma pospešiti.

Povsem jasno je, da se z uporabo obeh pravil v vezju nič ne spremeni. S pomočjo postopka presečnih množic lahko vsak graf poteka signalov (graf ne sme imeti zank brez zakasnitev in mora imeti homogeno



Slika 8.13: Uporaba pravila prenosa zakasnitev

zgradbo) spremenimo v sistolično procesorsko polje. Žal pa se v praksi kmalu pokaže, da je večina grafov poteka signalov zelo zapletenih in polna posebnosti, tako da je uporaba opisanega postopka lahko zelo kompliciranа in včasih ne daje želenega rezultata. Omeniti moramo, da se nemalokrat uporablja metoda, pri kateri iz grafa odvisnosti oblikujemo sistolično procesorsko polje brez vmesne preslikave na graf poteka signalov, s predpostavko, da je graf odvisnosti že kar graf poteka signalov.

Povezava vektorja s z metodo presečnih množic

Kot smo ugotovili v prejšnjih poglavjih, je preslikava iz grafa odvisnosti na graf poteka signalov odvisna od vektorja **s** (ta je pravokoten na hiperravnine grafa odvisnosti) in vektorja **d** (smer projekcije grafa odvisnosti). Dokazano je, da lahko z določeno izbiro vektorja **s** pri danem vektorju **d** (ta je enak kot pri običajni preslikavi, za katero smo se odločili) dosežemo povsem isti rezultat kot pri uporabi postopka presečnih množic. Od tu je razvidna tudi povezava med neposredno preslikavo iz grafa odvisnosti na sistolično procesorsako polje in običajno preslikavo na graf poteka signalov ter uporabo postopka presečnih množic.

8.4.3 Primer množenja matrik

Množenje dveh matrik matematično izrazimo kot $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, oziroma kot

$$c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}.$$

Ugotovimo, da lahko vsa množenja izračunamo istočasno, saj so med seboj popolnoma neodvisna. Gornjo enačbo moramo prirediti v obliko, ki vključuje enkratno prireditev spremenljivk (poglavje 8.3.2)

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} + a_{ik} b_{kj},$$

oziroma

$$\texttt{for}\ i = 1\ \texttt{to}\ N_1$$

$$\texttt{for}\ j = 1\ \texttt{to}\ N_2$$

$$\texttt{for}\ k = 1\ \texttt{to}\ N_3$$

$$c(i,j,k) = c(i,j,k-1) + a(i,k) * b(j,k),$$

kjer je k časovni indeks ali indeks rekurzije. Takšen algoritem je podan v globalni obliki, saj je na primer element a_{ik} oddan vsem vozliščem, ki imajo ista indeksa i in k. Lokaliziran algoritem izgleda takole:

$$\texttt{for}\ i = 1\ \texttt{to}\ N_1$$

$$\texttt{for}\ j = 1\ \texttt{to}\ N_2$$

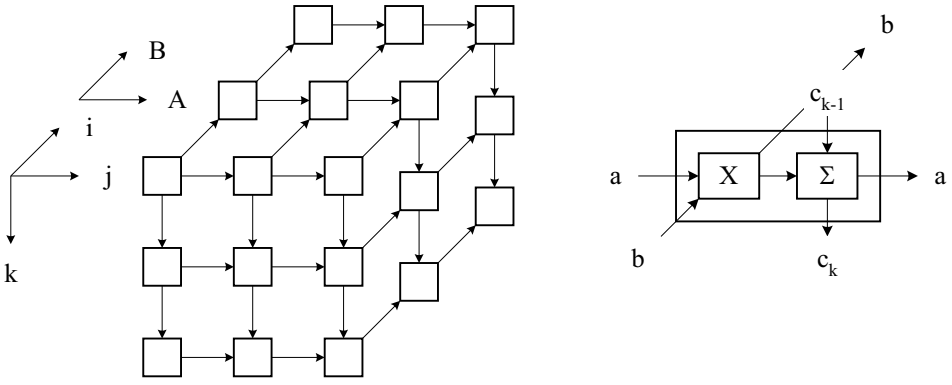
$$\texttt{for } k = 1 \texttt{ to } N_3$$

$$a(i,j,k) = a(i,j-1,k)$$

$$b(i,j,k) = b(i-1,j,k)$$

$$c(i,j,k) = c(i,j,k-1) + a(i,j,k) * b(i,j,k).$$

Na sliki 8.14 je prikazan graf odvisnosti, ki je trodimenzionalen, prikazane pa so tudi operacije v vozliščih. Projekcija grafa odvisnosti v smeri $(0,0,1)$ da graf poteka signalov, kot ga prikazuje slika 8.15. Ker ta graf nima oblike, ki bi zagotavljala sistolično strukturo, uporabimo postopek presečnim množic, in sicer pravilo prenosa zakasnitev. Presečne množice so prikazane na sliki 8.13. Po tem pravilu lahko vsem povezavam določene presečne množice dodamo zakasnitev, moramo pa jo dodati tudi na vhodne nize. Dobimo sistolični graf poteka signalov, kot ga prikazuje slika 8.16. V tem grafu ima vsaka povezava eno zakasnitev, tako da graf predstavlja že kar sistolično procesorsko polje.



Slika 8.14: Graf odvisnosti množenja matrik

8.4.4 Kvaliteta sistoličnega procesorskega polja

Običajno obstaja več faktorjev, ki opredeljujejo kvaliteto sistoličnega procesorskega polja, zato naštejmo najpomembnejše:

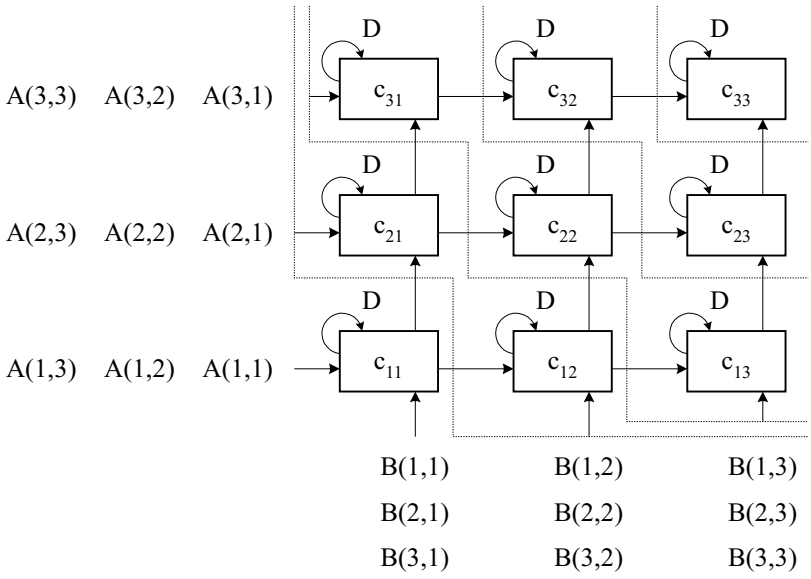
- čas izračuna (časovni interval med prvo in zadnjo izvršeno operacijo v procesorskem polju) in
- cevovodna (pipelining) perioda (časovni interval med dvem operacijama v procesorskem elementu), ki jo označujemo z α; njena recipročna vrednosti podaja izkoristek procesorskega elementa,
- število procesorskih elementov v polju,
- število vhodno/izhodnih povezav procesorskega polja.

Minimalni čas izračuna lahko pri danih vektorjih $\mathbf{s}$ in $\mathbf{d}$ izrazimo kot

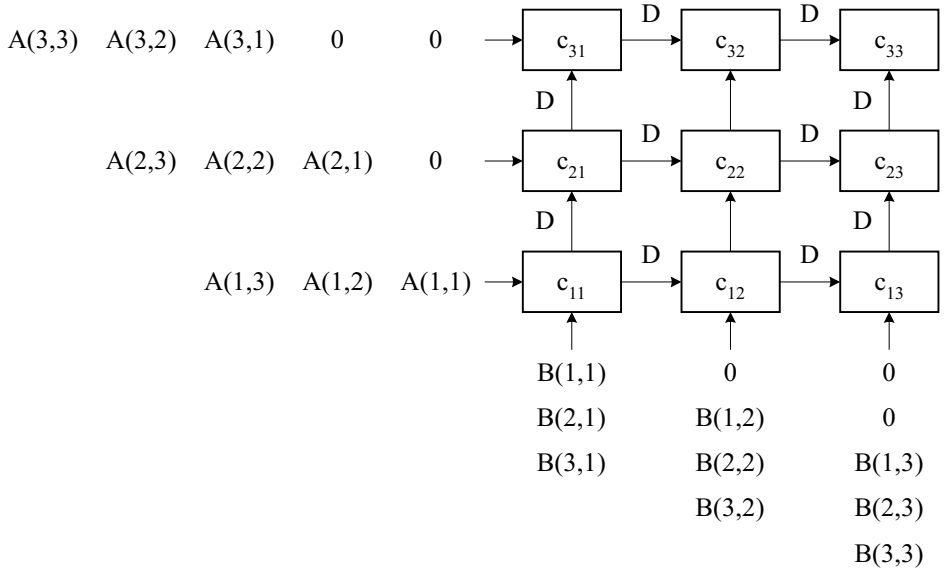
$$T = max(\mathbf{s}^{\mathrm{T}}(\mathbf{p} - \mathbf{q})) + 1,$$

kjer sta $\mathbf{p}$ in $\mathbf{q}$ vektorja poljubnega vozlišča v grafu odvisnosti. Gornjo enačbo rešujemo s pomočjo celoštevilskega programiranja, in sicer jo minimiziramo glede na vektor $\mathbf{s}$. Tudi cevovodna perioda naj bo čimmanjša (naboljše ena), izrazimo jo z enačbo

$$\alpha = \mathbf{s}^{\mathrm{T}}(\mathbf{p} + \mathbf{d}) - \mathbf{s}^{\mathrm{T}}\mathbf{p} = \mathbf{s}^{\mathrm{T}}\mathbf{d},$$



Slika 8.15: Graf poteka signalov množenja matrik z vrisanimi presečnimi množicami



Slika 8.16: Sistolično procesorsko polje množenja matrik

ki jo prav tako minimiziramo s pomočjo metode celoštevilskega programiranja. $\mathbf{p}$ je vektor poljubnega vozlišča v grafu odvisnosti, $\mathbf{d}$ je vektor projekcije, tako da vektorja $\mathbf{p}$ in $\mathbf{p} + \mathbf{d}$ predstavljata zaporedni vozlišči v grafu odvisnosti, ki se bosta preslikali v eno vozlišče grafa poteka signalov. Kot vidimo si optimizaciji časa izračuna in cevovodne periode nasprotujeta, tako da je težko poiskati zadovoljivo rešitev.

8.4.5 Ostale možnosti pri izvedbi sistoličnega procesorskega polja

Pri dosedanji obravnavi smo se omejili le na polja, ki so kot osnovni informacijski gradnik uporabljala besedo oziroma byte, lahko pa izdelamo procesorsko polje na nivoju bitov in v kasnejši izvedbi kombiniramo obe polji tako, da procesorsko polje na nivoju bitov predstavlja procesorski element v sistoličnem procesorskem polju na nivoju besed. Operacije v različnih procesorskih elementih potekajo

različno dolgo, zato mora biti ura prilagojena najpočasnejšemu procesorskemu elementu. Kot rešitev uporabimo dodatno, hitrejšo uro, za najpočasnejše procesorske elemente. Govorimo o "multirate" sistoličnem procesorskem polju. Kadar je cevovodna perioda večja od ena, lahko v času praznega vhoda uporabimo neko drugo vhodno sekvenco, tako da rešujemo dva neodvisna problema hkrati na istem sistoličnem procesorskem polju. Govorimo o "pipeline interleaving" sistoličnem procesorskem polju.

Za zaključek omenimo, da so sistolična procesorska polja primerna za reševanje problemov dinamičnega programiranja, nevronskih sistemov itd.

8.5 Množenje matrike z vektorjem

8.5.1 Osnovni matrični algoritmi

Osnovni problemi matričnega računa se pojavljajo v množici numeričnih, kot nenumeričnih algoritmih. Primeri se vrste od reševanja sistemov enačb, do predstavitve grafov. V tem poglavju si bomo ogledali, kako lahko operacije na matrikah izvajamo tudi vzporedno.

Osnovni postopek preslikave algoritma na sistoločna polja, znan iz dodedanjih poglavij si bomo ogledali na enostavnem primeru množenja matrike z vektorjem. Za isti primer nameravamo prikazati dva postopka reševanja istega problema. V prvem primeru s prireditvijo indeksnega prostora, naletimo na problem reševanja sistema "affinih" enačb, v drugem primeru pa se z ustrezno prireditvijo spremenljivke temu problemu izognemo.

8.5.2 Predstavitev in pretvorba algoritma v sistem enoličnih rekurzivnih enačb

Katerikoli matematični postopek moremo vedno podati v različnih oblikah. Pri tem uporabljamo različne zapise, vse od formalnega zapisa, preprostega opisa postopka do algoritmov za izračun različnih oblik. Kot smo spoznali, je za preslikavo postopkov na vzporedna procesorska polja posebej primeren zapis v obliki sistema enoličnih rekurzivnih enačb.

Formalni zapis algoritma

Množenje matrike z vektorjem je operacija, ki jo formalno zapišemo z izrazom

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x}.$$

V izrazu nastopajo tri spremenljivke. Parametra operacije sta matrika $\mathbf{A}$, v splošnem dimenzij $(m \times n)$ in vektor $\mathbf{x}$ z n elementi. Rezultat je vektor $\mathbf{y}$, ki ima pri podanih dimenzijah m elementov. Zaradi poenostavitve vzemimo, da je matrika $\mathbf{A}$ kvadratna, njene dimenzije pa so $(n \times n)$.

Iterativna rešitev

Kakšen je rezultat zapisane operacije? Zapis v obliki končne vsote produktov elementov $a_{ij} \subset \mathbf{A}$ in $x_j \subset \mathbf{x}$,

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \qquad i = 1, \ldots, n,$$

je za matematika dovolj preprost in dovolj nedvoumen.

To je zapis v iterativni obliki. Če si ogledamo splošni iterativni zapis algoritma,

$$a_j(\mathbf{q}) := iteracija(*, argument, območje),$$

vidimo, da je v našem primeru uporabljena aditivna operacija $*$ $(\sum)$, *argumenta* operacije sta a_{ij} in x_j, *območje* parametrov pa obsega števila iz množice $[1, \ldots, n]$.

Preslikava v rekurzivno obliko

Algoritem v rekurzivni obliki zapišemo z izrazom

$$a_j(\mathbf{q}) := a_j(\mathbf{q}) * argument$$

pri čemer ponovimo operacijo za vsako vrednost $s \in$ *območja*.

V nadaljevanju pišemo parametre spremenljivk v obliki indeksnih točk. V našem primeru, ob vpeljavi ustreznih zank za določanje vrednosti parametrov in ob uporabi ustreznih začetnih vrednosti za y_i (nevtralni element za seštevanje je enak 0), dobimo naslednji rekurzivni algoritem:

```
for i := 1 to n do
  y(i) := 0
  for j := 1 to n do
    y(i) := y(i) + a(i,j)x(j)
  end
end
```

Rezultat smo dobili z uporabo postopka A. Ker v iterativnem zapisu ni rekurzivne uporabe argumentov (y ne nastopa kot argument v iterativnem zapisu), uporaba postopka C ni potrebna.

Dopolnjevanje nepopolnih indeksov

Kot vidimo, indeksni prostor, uporabljen pri spremenljivkah y in x, ni popoln, saj ne obsega vseh indeksov algoritma. Algoritem dopolnimo do polne indeksne oblike z uporabo postopka D. Indeksni prostor funkcije za določanje indeksov $\mathbf{q} = \varphi_{ij}(\mathbf{p})$ razširimo s prenosom obstoječih indeksov in s prireditvijo vrednosti 0 za manjkajoči indeks.

$$y(i) \rightarrow y(i,0)$$
$$x(j) \rightarrow x(j,0)$$

Algoritem v popolni indeksni obliki je torej enak

```
for i := 1 to n do
  y(i,0) := 0
  for j := 1 to n do
    y(i,0) := y(i,0) + a(i,j)x(j,0)
  end
end
```

Določanje podatkovnih odvisnosti

Najprej zapišimo indeksno množico algoritma! V algoritmu nastopata indeksa i in j. Indeksna množica algoritma je enaka

$$J = \{\mathbf{p};\ \mathbf{p} = \begin{bmatrix} i \\ j \end{bmatrix}\}$$

pri čemer so meje indeksov določene z neenačbama

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{p} \geq \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \mathbf{p} \leq \begin{bmatrix} n & n \end{bmatrix}.$$

Na tej osnovi izračunamo podatkovne odvisnosti, posebej za spremenljivki y in x.

$$\mathbf{d}_y = \mathbf{p} - \varphi_y(\mathbf{p})$$
$$\mathbf{d}_y = \begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} i \\ 0 \end{bmatrix}$$
$$\mathbf{d}_y = \begin{bmatrix} 0 \\ j \end{bmatrix}$$

$$\mathbf{d}_x = \mathbf{p} - \varphi_x(\mathbf{p})$$
$$\mathbf{d}_x = \begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} j \\ 0 \end{bmatrix}$$
$$\mathbf{d}_x = \begin{bmatrix} i-j \\ j \end{bmatrix}.$$

Vidimo, da so podatkovne odvisnosti linearno odvisne od indeksov i in j. Takšne podatkovne odvisnosti imenujemo tudi "afine" podatkovne odvisnosti. Določimo koeficiente "afinih" podatkovnih odvisnosti!

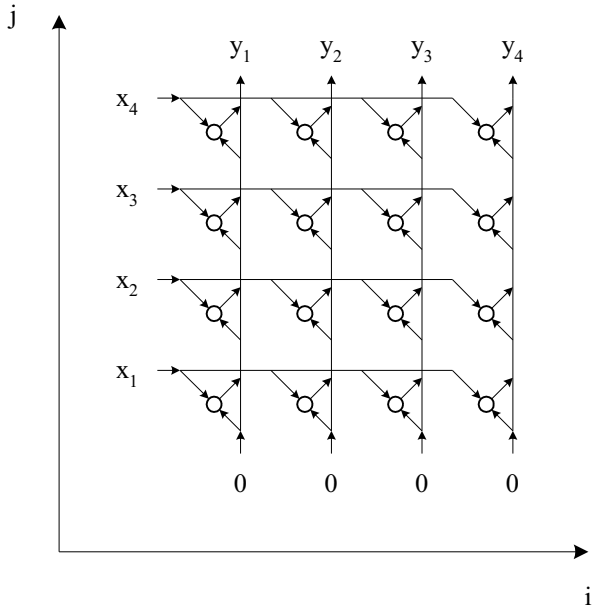
$$\mathbf{d}_y = \mathbf{A}_y\mathbf{p} + \mathbf{d}_0$$
$$\mathbf{d}_y = \begin{bmatrix} 0 \\ j \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$
$$\mathbf{A}_y = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$
$$\mathbf{d}_{0_y} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{d}_x = \mathbf{A}_x\mathbf{p} + \mathbf{d}_0$$
$$\mathbf{d}_x = \begin{bmatrix} i-j \\ j \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$
$$\mathbf{A}_x = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$
$$\mathbf{d}_{0_x} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Preoblikovanje globalnega v lokalno razširjanje

Po dopolnitvi indeksov vidimo, da v algoritmu za množenje matrike in vektorja nastopata tako izračunsko kot tudi podatkovno razširjanje. Globalno razširjanje v algoritmu prikazuje slika 8.17. Odpraviti želimo obe vrsti razširjanja in preoblikovati sistem "afinih" rekurzivnih enačb v sistem enoličnih rekurzivnih enačb. Pri tem si bomo pomagali z uporabo postopka E.

Preoblikovanje izračunskega razširjanja Izračunsko globalno razširjanje nastopa pri spremenljivki y. Da bi globalno razširjanje odpravili. Sledimo korakom:



Slika 8.17: Graf odvisnosti algoritma za množenje matrike in vektorja

1. Določimo matriko $\mathbf{B}_y$.

$$\mathbf{B}_y = \mathbf{I} - \mathbf{A}_y$$
$$\mathbf{B}_y = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$
$$\mathbf{B}_y = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

2. Izvajanje vrstičnih preslikav ni potrebno, saj matrika $\mathbf{B}_y$ v zadnji vrstici vsebuje le ničelne elemente.

3. Rang matrike $\mathbf{B}_y$ je enak 1, $rang(\mathbf{B}_y) = n - 1$. Vektor razširjanja $\mathbf{r}_y$ določimo na osnovi poddeterminant zadnje vstice matrike $\mathbf{B}_y$

$$\mathbf{r}_y = (b^*_{y1} b^*_{y2})^{\mathrm{T}}$$
$$b^*_{y1} = 0$$
$$b^*_{y2} = 1$$
$$\mathbf{r}_y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

4. Dejanski vektor razširjanja izberemo med vrednostima $\mathbf{r}_y$ in $-\mathbf{r}_y$.

Izračunsko globalno razširjanje v našem algoritmu zapišemo kot

$$y(i,0) := y(i,0) + a(i,j)x(j,0)$$
$$y(\varphi_{y0}(\mathbf{p})) := F(\ldots, y(\varphi_{y0}(\mathbf{p}))\ldots).$$

Izraz iz sistema rekurzivnih enačb zapišemo v ekvivalentni obliki kot

$$y(\mathbf{p}) := F(\ldots, y(\mathbf{p} - \mathbf{r}_y), \ldots)$$
$$y(i,j) := y(i,j-1) + a(i,j)x(j,0)$$

z začetnim pogojem

$$y(i,0) := 0$$

ali kot

$$y(\mathbf{p}) := F(\ldots, y(\mathbf{p} - (-\mathbf{r}_y)), \ldots)$$
$$y(i,j) := y(i,j+1) + a(i,j)x(j,0)$$

z začetnim pogojem

$$y(i,n+1) := 0.$$

Preoblikovanje podatkovnega razširjanja Globalno podatkovno razširjanje v algoritmu nastopa pri spremenljivki x. Tudi korake, potrebne za preslikavo algoritma z globalnim podatkovnim razširjanjem v algoritem z lokalnim podatkovnim razširjanjem najdemo v postopku E.

1. Določimo matriko $\mathbf{B}_x$.

$$\mathbf{B}_x = \mathbf{I} - \mathbf{A}_x$$
$$\mathbf{B}_x = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$$
$$\mathbf{B}_x = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

2. Izvajanje vrstičnih preslikav ni potrebno, saj ima matrika $\mathbf{B}_x$ v zadnji vrstici le ničelne elemente.

3. Rang matrike $\mathbf{B}_x$ je enak 1. Vektor razširjanja $\mathbf{r}_x$ sestavimo iz poddeterminant zadnje vrstice matrike $\mathbf{B}_x$

$$\mathbf{r}_x = (b^*_{x1} b^*_{x2})^{\mathrm{T}}$$
$$b^*_{x1} = 1$$
$$b^*_{x2} = 0$$
$$\mathbf{r}_x = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

4. Za vektor razširjanja smemo izbrati vrednost $\mathbf{r}_x$ ali $-\mathbf{r}_x$.

Tudi globalno podatkovno razširjanje opravimo na podoben način kot globalno izračunsko razširjanje podatkov. Po potrebi dodamo novo rekurzijsko enačbo, ki zagotavlja lokalno razširjanje podatkov. Lokalno razširjanje podatkov nam v opisanem primeru zagotavljata enačba

$$x(\mathbf{p}) := x(\mathbf{p} - \mathbf{r}_x)$$

$$x(i,j) := x(i-1,j)$$

z začetnim pogojem

$$x(0,j) := x_j$$

kot tudi enačba

$$x(\mathbf{p}) := x(\mathbf{p} - (-\mathbf{r}_x))$$

$$x(i,j) := x(i+1,j)$$

z začetnim pogojem

$$x(n+1,j) := x_j.$$

Glede na izbiro vektorja razširjanja spremenljivke y za osnovno rekurzivno enačbo uporabimo eno od enačb

$$y(i,j) := y(i,j-1) + a(i,j)x(i,j)$$

ali

$$y(i,j) := y(i,j+1) + a(i,j)x(i,j)$$

z že znanimi začetnimi pogoji.

Izbira preslikave

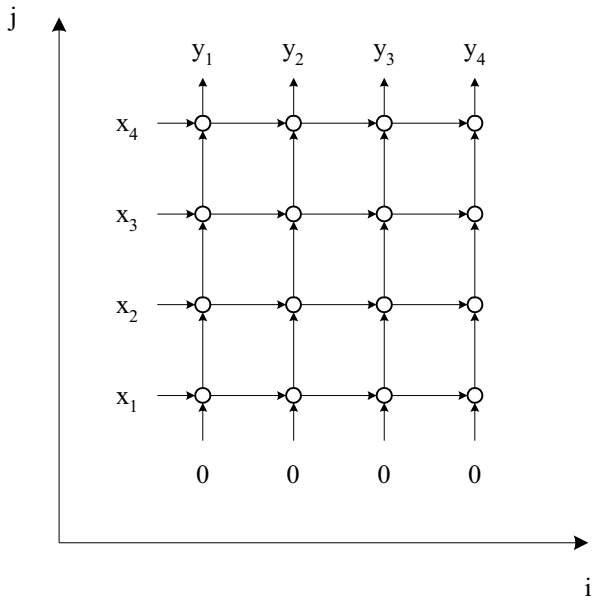
Kot vidimo, smo algoritem za množenje matrike z vektorjem zapisali s štirimi različnimi kombinacijami enoličnih rekurzivnih enačb. Med njimi izberimo prvo možnost,

```
for i := 1 to n
  y(i,0) := 0
for j := 1 to n
  x(0,j) := x(j)
for i := 1 to n
  for j := 1 to n
    x(i,j) := x(i-1,j)
    y(i,j) := y(i,j-1) + a(i,j)x(i,j)
```

Izračunajmo še podatkovne odvisnosti

$$\mathbf{d}_x = \begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} i-j \\ j \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$\mathbf{d}_y = \begin{bmatrix} i \\ j \end{bmatrix} - \begin{bmatrix} i \\ j-1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Podatkovne odvisnosti so torej konstantne, dobili smo sistem enoličnih rekurzivnih enačb. Na sliki 8.18 si oglejmo le še graf podatkovnih odvisnosti dobljenega algoritma z lokalnim razširjanjem.



Slika 8.18: Graf odvisnosti algoritma za množenje matrike in vektorja

8.5.3 Preslikava sistema rekurzivnih enačb na sistoločno polje

Naša naslednja naloga je prenos sistema enoličnih rekurzivnih enačb na sistolično polje. Uporabili bomo linearno sistolično polje z možnostjo pretoka podatkov v obe smeri. Pri izbiri ustrezne preslikave si bomo pomagali z uporabo postopka na sliki ??. Sledimo korakom, predstavljenim na sliki ??!

K1: Za osnovo sinteze vzamemo sistem enoličnih rekurzivnih enačb, ki smo ga dobili z uporabo postopkov A, C, D in E.

K3: Podatkovne odvisnosti sistema enoličnih rekurzivnih enačb so konstantne, zato nadaljujemo s korakom K4.

K4, K5: Določimo najprej časovni del matrike za preslikavo $\mathbf{T}$. Za osnovo vzamemo neenačbo

$$\pi \times \mathbf{d}_i > 0.$$

Veljati mora torej

$$\pi \times \mathbf{d}_x = \pi \times \begin{bmatrix} 1 \\ 0 \end{bmatrix} > 0$$
$$\pi \times \mathbf{d}_y = \pi \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} > 0,$$

torej je

$$\pi = \begin{bmatrix} 1 & 1 \end{bmatrix}.$$

Izbrana vrednost minimizira čas izvajanja algoritma.

K6, K7, K8, K9, K10: Povezovalna matrika dvosmernega linearnega polja je enaka

$$\mathbf{P} = \begin{bmatrix} 1 & -1 \end{bmatrix}.$$

Matrika podatkovnih odvisnosti algoritma pa je

$$\mathbf{D} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

Izbrati moramo povezovalne korene, ki jih bomo uporabili v vzporednem algoritmu. Pri tem moramo upoštevati pogoj

$$\sum_j \chi_{ji} \leq \sum_j d_{ji}^k$$

in to, da mora biti dobljena matrika za preslikavo

$$\mathbf{T} = \begin{bmatrix} \pi \\ S \end{bmatrix}$$

nesingularna. Krajevni del preslikave je rešitev matrične enačbe

$$\mathbf{S} \times \mathbf{D} = \mathbf{P} \times \mathbf{X}.$$

Med možnimi rešitvami si oglejmo dve,

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

in

$$\mathbf{T}_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}.$$

Obe preslikavi si podrobneje oglejmo!
Preslikava

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

preslika posamezne indeksne točke algoritma v časovnih trenutkih na procesorske elemente tako, kot prikazuje tabela ??. Podatkovne odvisnosti se v časovni prostor preslikajo kot

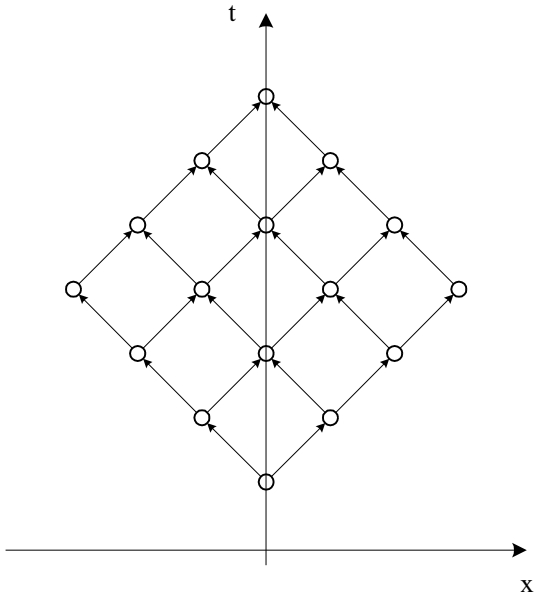
$$d_x^k = \pi \cdot \mathbf{d}_x = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$d_y^k = \pi \cdot \mathbf{d}_y = \begin{bmatrix} 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

v krajevnem prostoru pa se za spremenljivki x in y oblikujeta povezavi

$$d_x^M = \mathbf{S} \cdot \mathbf{d}_x = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$$

$$d_y^M = \mathbf{S} \cdot \mathbf{d}_y = \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = -1$$

Če privzamemo, da pozitivne vrednosti predstavljajo pomik v desno, vidimo, da vrednosti x v linearnem polju potujejo v desni, vrednosti y pa v levi smeri. Časovne aktivnosti na posameznih procesorskih elementih prikazuje slika 8.19.



Slika 8.19: Časovno-krajevni graf algoritma za sistolično polje ($\mathbf{T}_1, n = 4$)

Uporabljeno procesorsko polje si oglejmo na sliki 8.19.

Za izvedbo v tej obliki potrebujemo $2n - 1$ procesorskih elementov, skupni čas izvajanja algoritma pa je enak

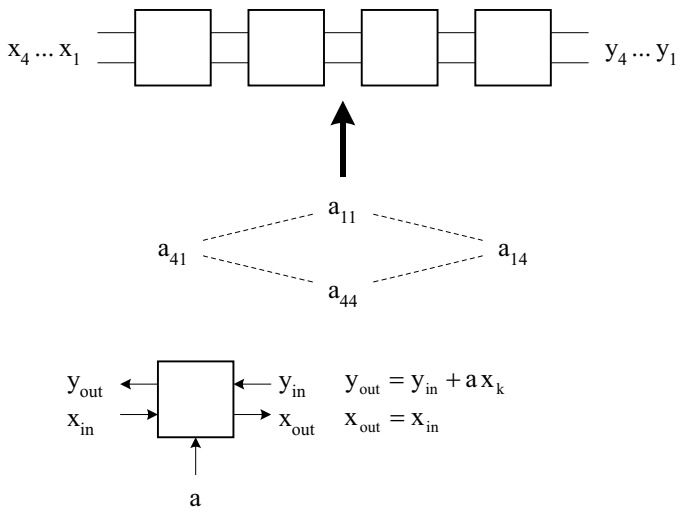
$$t = \frac{max\ \pi(\mathbf{p}_v - \mathbf{p}_w) + 1}{min\ \pi \cdot \mathbf{d}_i} = 2n - 1.$$

S preslikavo

$$\mathbf{T}_2 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

se posamezne indeksne točke algoritma preslikajo v prostor sistoličnega polja tako, kot to prikazuje tabela ??. Časovna preslikava je enaka preslikavi iz T_1, zato so tudi rezultati identični. Slika v krajevnem prostoru pokaže, da za spremenljivki x in y dobimo povezavi

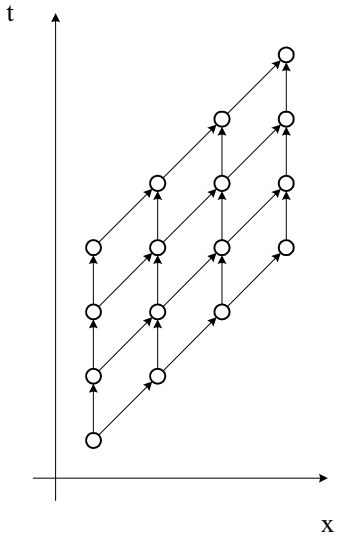
$$d_x^M = \mathbf{S} \cdot \mathbf{d}_x = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0$$



Slika 8.20: Dvosmerno procesorsko polje ($\mathbf{T}_1, n = 4$)

$$d_y^M = \mathbf{S} \cdot \mathbf{d}_y = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1.$$

Vrednosti x v linearnem polju so v tem primeru stacionirane na posameznih elementih polja, vrednosti y pa se pomikajo v desni smeri. Algoritem je torej enosmeren. Časovne aktivnosti na posameznih procesorskih elementih prikazuje slika 8.21.

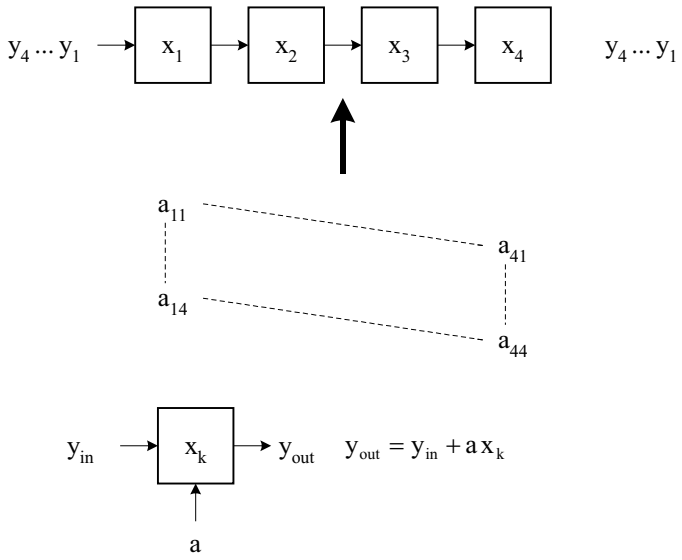


Slika 8.21: Časovno-krajevni graf enosmernega algoritma ($\mathbf{T}_2, n = 4$)

Za izvedbo v tej obliki potrebujemo n procesorskih elementov. Procesorsko polje je prikazano na sliki 8.22. Skupni čas izvajanja algoritma je enak kot pri preslikavi $\mathbf{T}_1$, $t = 2n - 1$.

8.5.4 Transpozicija matrike

Naj bo $\mathbf{A} = \in R^{n \times n}$. Poiskati želimo transponse matrike $\mathbf{A}$. Matrika $\mathbf{A}$ je določena kot:



Slika 8.22: Enosmerno procesorsko polje ($\mathbf{T}_2, n = 4$)

Matrika odvisnosti $\mathbf{D}$ je:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}.$$

V transponirani matriki $\mathbf{A}^{\mathrm{T}}$ je vsaka vrstica osnovne matrike $\mathbf{A}$ stolpec. Elementi matrike so podatkovni objekti in lahko vsebujejo pojubne vrednosti.

Algoritem (Transpozicija matrike)

Sekvenčni postopek lahko prikažemo z enostavnim algoritmom, katerega rezultat je transponirana matrika $\mathbf{A}^{\mathrm{T}}$.

```
for i := 2 to n do
  for j := 1 to i − 1 do
    a(i, j) := a(j, i)
  end
end
```

Algoritem, zapisan v obliki enkratne prireditve, zahteva $O(n^2)$ operacij. Zanimiva je matrika odvisnosti $\mathbf{D}$, ki jo določimo kot medsebojno odvisnost spremenljivke $a(i, j)$ in $a(j, i)$. Ta je:

$$\begin{bmatrix} i-j \\ j-i \end{bmatrix}.$$

V tem primeru vidimo, da se podatki globalno razširjajo. Zato rešitev ni enostavna. V celotnem postopku moramo upoštevati postopek odpravljanja globalnega razširjanja.

8.5.5 Množenje matrike z vektorjem

Kot predhodno je tudi sedaj matrika $\mathbf{A}$ dimenzije $n \times n$. Po podanem algoritmu

```
for i := 1 to n do
  y(i) := 0
  for j := 1 to n do
    y(i) := y(i, j − 1) + a(i, j)x(i, j)
  end
end
```

iterativno izračunamo produkt matrike z vektorjem $\mathbf{y} = \mathbf{A} \cdot \mathbf{x}$. V algoritem je vgrajen izraz, ki omogoča zaporedno računanje komponent vektorja $\mathbf{y}$ z upoštevanjem izraza

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

Po postopku dopolnitve indeksov, ko se $y(i)$ preslika v $y(i, 0)$ in $x(j)$ v $x(0, j)$, dobimo naslednji algoritem, zapisan v obliki enkratne prireditve:

```
for i := 1 to n do
  y(i) := 0
  for j := 1 to n do
  x(i, j) = x(i − 1, j)
    y(i, j) := y(i, j − 1) + a(i, j)x(i, j)
  end
end
```

Med spremenljivkami $a(i, j)$ ni medsebojnih odvisnosti oziroma velja $\mathbf{d}_{aa} = [0\ \ 0]$. Spremenljivka $b(i, j)$ je neposredno odvisna od spremenljivke $b(i-1, j)$, torej $\mathbf{d}_{bb} = [1\ \ 0]$. Spremenljivka $c(i, j)$ pa je neposredno odvisna od $c(i, j-1)$, $a(i, j)$ in $b(i, j)$. Neničelni vektor odvisnosti je $\mathbf{d}_{cc} = [0\ \ 1]$. Matrika odvisnosti $\mathbf{D}$ je :

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Po metodi podatkovnih odvisnosti poiščemo najprej časovno transformacijo π. Možna rešitev je $\pi = [1\ 1\ 1]^{\mathrm{T}}$. Čeprav velja $\pi \cdot \mathbf{d}_{aa} = 0$, to pomeni le, da vrednosti $a(i, j)$ ne bodo potovale po procesorskem polju, kar se ujema z dejstvom, da med seboj niso odvisne. Pri naslednjem koraku moramo poiskati matriko $\mathbf{S}$, ki zadošča enačbi

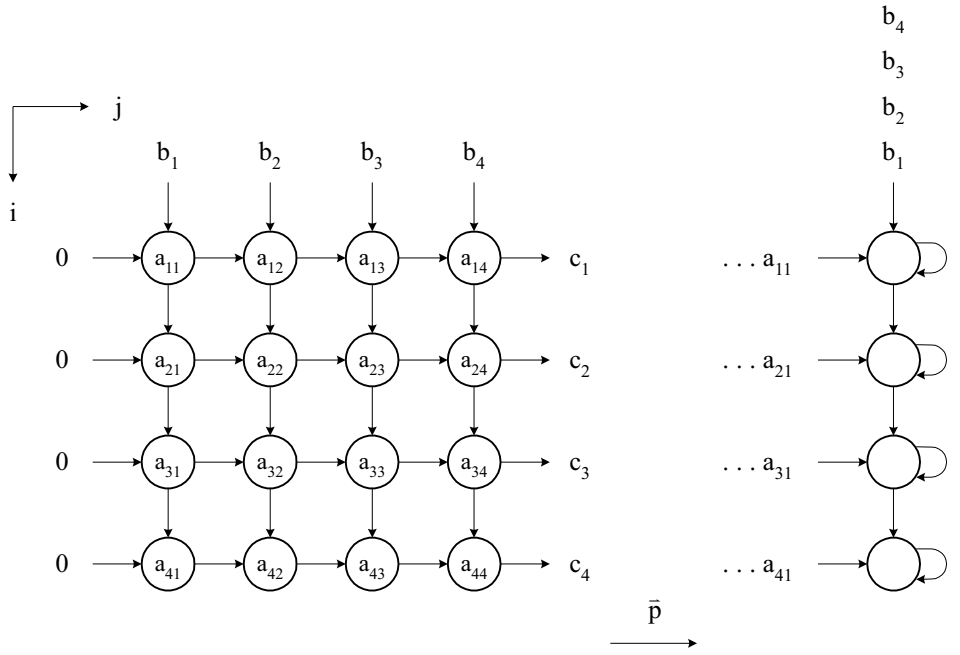
$$\mathbf{SD} = \mathbf{PK}.$$

Naj bo matrika povezav v procesorskem polju $\mathbf{P} = [0\ \ 1]$. Možna rešitev za $\mathbf{K}$ je

$$\mathbf{K} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Pripadajoča rešitev za $\mathbf{S}$ je

$$\begin{bmatrix} 1 & 0 \end{bmatrix}.$$



Slika 8.23: Graf odvisnosti in graf poteka signalov za množenje matrike z vektorjem ($n = 4$)

Sistolični algoritem želimo poiskati še z grafično metodo. V ta namen najprej določimo graf odvisnosti, ki pripada algoritmu (slika 8.23.

Naj bo smer projekcije $\mathbf{p} = [0 \;\; 1]$. Izberimo osnovno razvrščanje ter tako matriko smeri v procesorskem polju, da je $\mathbf{P}^{\mathrm{T}}\mathbf{p} = 0$

$$\mathbf{P}^{\mathrm{T}} \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

Določimo preslikavo vozlišč, povezav ter vhodnih in izhodnih vozlišč.

Preslikava vozlišč:

$$\begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = i.$$

Preslikava povezav:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

Za spremenljivke a_{ij}, med katerimi ni medsebojnih odvisnosti, velja enako tudi po preslikavi. Odvisnost med b_j oziroma povezava v smeri $\mathbf{d}_{bb} = [1 \;\; 0]$, se preslika v povezavo v smeri 1, časovne zakasnitve ni. Spremenljivke c_{ij} pa postaneja medsebojno neodvisne. Dobimo "povezavo" v smeri 0 ali z drugimi besedami, spremenljivka je zapisana v vozlišču i, zakasnitev računanja meri eno časovno enoto.

Preslikava vhodnih vozlišč:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i & 1 \\ j & j \end{bmatrix} = \begin{bmatrix} j & j \\ i & 1 \end{bmatrix}.$$

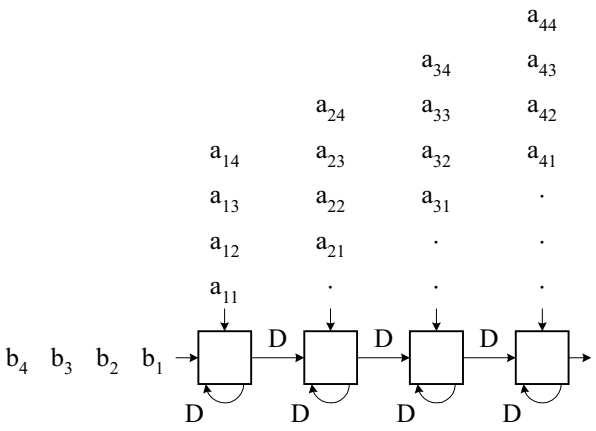
Izračuni, katerih vhodni podatki so spremenljivke a_{ij}, se bodo preslikali na vozlišča i v GPS, čas, ko se bodo izvajali, pa bo $t = j$. Izračuni, katerih vhodni podatki so spremenljivke b_j, pa na vozlišče na mestu 1, čas izvajanja bo prav tako $t = j$.

Preslikava izhodnih vozlišč:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ n \end{bmatrix} = \begin{bmatrix} n \\ i \end{bmatrix}.$$

Izračuni, katerih rezultati so vrednosti c_{ij}, se bodo preslikali na vozlišča i, čas njihovega izvajanja bo $t = n$.

Graf poteka signalov je na sliki 8.23. Z uporabo časovnih pravil dobimo linearno sistolično polje velikosti n (slika 8.24).



Slika 8.24: Sistolično polje za množenje matrike z vektorjem ($n = 4$)

Procesor P_j začne računati v času $t = j$ in potem izvaja operacije oblike $c = c + ab$, pri čemer je a vhodni podatek in pomeni ustrezen element j-te vrstice matrike $\mathbf{A}$, b pa ustrezen element vektorja $\mathbf{b}$. Hkrati pa mora element b poslati svojemu desnemu sosedu.

Celoten algoritem zahteva $2n - 1$ časovnih korakov, saj procesor P_n izračuna element b_n v času $t = 2n - 1$.

8.5.6 Množenje matrik

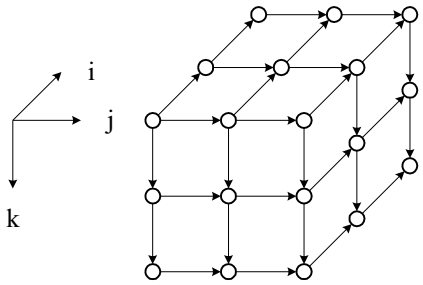
Zaporedni algoritem za množenje dveh pravokotnih matrik

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B},$$

pri čemer velja $\mathbf{A} \in R^{n \times r}$, $\mathbf{B} \in R^{r \times m}$ in $\mathbf{C} \in R^{n \times m}$, je

```
for i := 1 to n
  for j := 1 to m
    for k := 1 to r
      c_ij := c_ij + a_ik b_kj.
```

Na tak način zapisan algoritem implicitno vključuje globalno razširjanje podatkov med procesorji, saj mora biti na primer element a_{ik} prenesen do vseh indeksnih točk z enakima indeksoma i in k. Da se izognemo globalnemu razširjanju, zapišemo algoritem v obliki enkratne prireditve.



Slika 8.25: Graf odvisnostie za množenje matrik

```
for i := 1 to n
  for j := 1 to m
    for k := 1 to r
      a(i,j,k) := a(i,j-1,k)
      b(i,j,k) := b(i-1,j,k)
      c(i,j,k) := c(i,j,k-1) + a(i,j,k)b(i,j,k)
```

Med podatki v izračunih tega algoritma so tri smeri odvisnosti: $\mathbf{d}_{aa} = [0\ \ 1\ \ 0]$, $\mathbf{d}_{bb} = [1\ \ 0\ \ 0]$ in $\mathbf{d}_{cc} = [0\ \ 0\ \ 1]$. Matrika odvisnosti, ki algoritmu pripada, je:

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Graf odvisnosti, ki pripada temu algoritmu v primeru, ko velja $n = m = r = 3$, vidimo na sliki 8.25. Po metodi podatkovnih odvisnosti poiščimo najprej časovno transformacijo π. Ena možnih rešitev je $\pi = [1\ \ 1\ \ 1]^{\mathrm{T}}$. Primer matrike povezav v procesorskem polju je

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

in možna rešitev za $\mathbf{K}$ je

$$\mathbf{K} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

V procesorskem polju potujejo elementi matrike $\mathbf{A}$ v smeri $[0\ \ 1]$, elementi matrike $\mathbf{B}$ v smeri $[1\ \ 0]$, elementi matrike $\mathbf{C}$ pa so zapisani v procesorjih.

Pripadajoča rešitev za $\mathbf{S}$, ki zadošča enačbi $\mathbf{SD} = \mathbf{PK}$, je

$$\mathbf{S} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Sistolični algoritem poiščimo še z grafično metodo. Za smer projekcije grafa odvisnosti izberimo $\mathbf{p} = [0\ \ 0\ \ 1]^{\mathrm{T}}$ in predpostavimo osnovno razvrščanje, $\mathbf{s} = [0\ \ 0\ \ 1]^{\mathrm{T}}$, ter matriko povezav v sistoličnem polju:

$$\mathbf{P}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Določimo še vse tri preslikave):
Preslikava vozlišč:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i \\ j \end{bmatrix}.$$

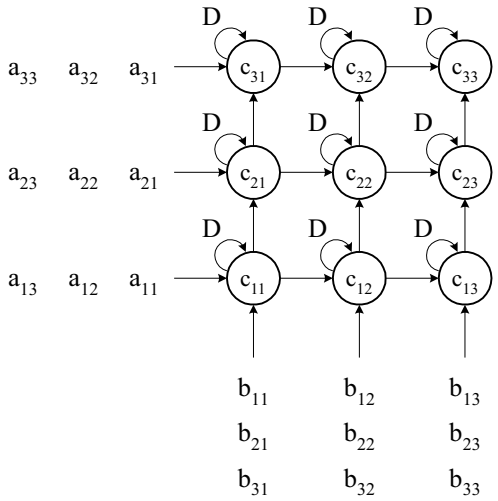
Preslikava povezav:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Preslikava vhodnih vozlišč:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i & 1 \\ 1 & j \\ k & k \end{bmatrix} = \begin{bmatrix} k & k \\ i & 1 \\ 1 & j \end{bmatrix}.$$

Elementi a_{ik} matrike $\mathbf{A}$, ki vstopajo v vozlišča grafa odvisnosti na mestu (i, j, k), v grafu poteka signalov vstopajo v vozlišča na mestu $(i, 1)$ v času $t = k$. Elementi b_{kj} matrike $\mathbf{B}$, ki vstopajo v grafu odvisnosti v vozlišča na mestu $(1, j, k)$, pa v grafu poteka signalov vstopajo v vozlišča na mestu $(1, j)$ prav tako v času $t = k$.



Slika 8.26: Graf poteka signalov za množenje matrik

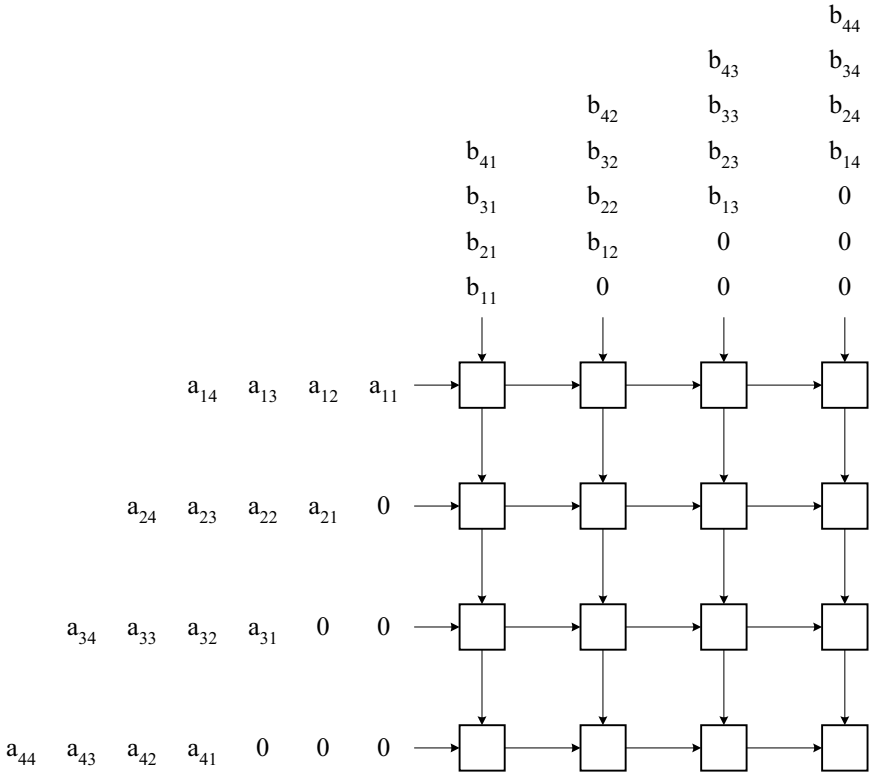
Preslikava izhodnih vozlišč:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ n \end{bmatrix} = \begin{bmatrix} n \\ i \\ j \end{bmatrix}.$$

Elementi c_{ik} matrike $\mathbf{C}$, ki so rezultati izračunov, katere predstavljajo v grafu odvisnosti vozlišča na mestu $(1, j, n)$, po preslikavi postanejo izhodni podatki vozlišč grafa poteka signalov na mestu $(1, j)$ v času $t = n$.

Rezultat te preslikave je graf poteka signalov na sliki 8.26. Zakasnitev pomeni večanje indeksa k in označuje odvisnost v smeri [0 0 1].

Da iz grafa poteka signalov dobimo sistolično polje, moramo uporabiti časovna pravila in poskrbeti, da podatki prihajajo z ustrezno zakasnitvijo. Rezultat je pravokotno sistolično polje na sliki 8.27.



Slika 8.27: Sistolično polje za množenje matrik

8.5.7 Množenje pasovnih matrik

Posebno strukturo matrik lahko upoštevamo tudi v algoritmu. Tako na primer za množenje pasovnih matrik dobimo drugačen sistolični algoritem.

Naj bosta $\mathbf{A}$ in $\mathbf{B}$ pasovni matriki velikosti $n \times n$ z ustreznima širinama pasov p in q, pri čemer je $n \gg p, q$. Procesorsko polje velikosti $n \times n$ je neučinkovito, saj za vzporedno rešitev problema zadošča polje velikosti $p \times q$.

Odvisnosti med podatki ostanejo take kot pri množenju navadnih matrik. Graf odvisnosti (slika 8.28) se spremeni, ker upoštevamo le neničelne izračune.

Sistolični algoritem poiščemo z grafično metodo. Za smer projekcije izberemo $\mathbf{p} = [1\ 1\ 1]^{\mathrm{T}}$, vektor razvrščanja pa naj bo $\mathbf{s} = [0\ 0\ 1]^{\mathrm{T}}$. Taka matrika povezav v sistoličnem polju, da velja $\mathbf{P}^{\mathrm{T}}\mathbf{p} = 0$, je

$$\mathbf{P}^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}.$$

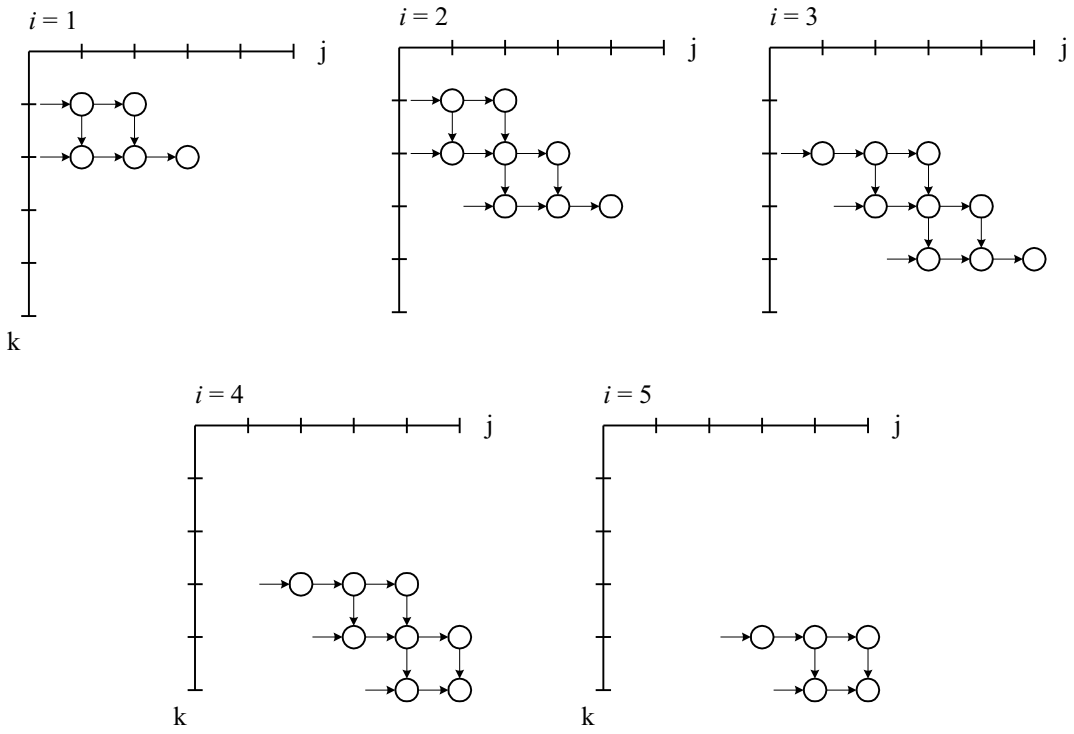
Določimo, kako se preslikajo vozlišča in povezave grafa odvisnosti.

Preslikava vozlišč:

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} = \begin{bmatrix} i-k \\ j-k \end{bmatrix}.$$

Preslikava povezav:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}.$$



Slika 8.28: Graf odvisnosti za množenje pasovnih matrik ($n = 5, p = q = 3$) - povezave, ki tečejo v smeri [1 0 0], niso narisane

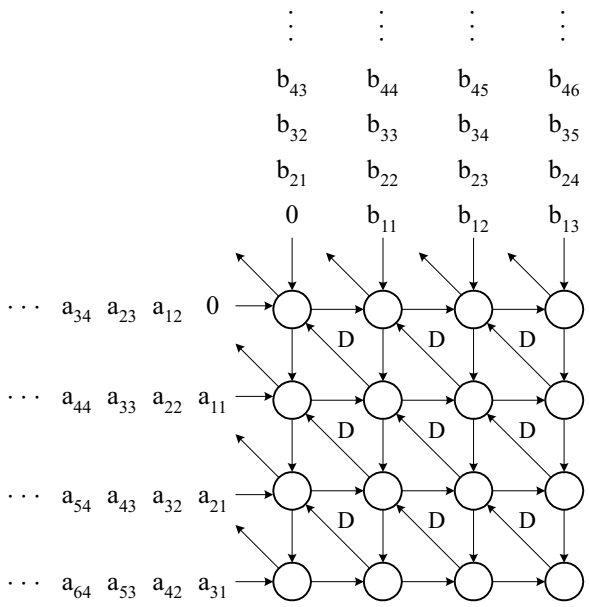
Preslikava vhodnih vozlišč:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} i & 1 \\ 1 & j \\ k & k \end{bmatrix} = \begin{bmatrix} k & k \\ i-k & 1-k \\ 1-k & j-k \end{bmatrix}.$$

Preslikava izhodnih vozlišč:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \\ w \end{bmatrix} = \begin{bmatrix} w \\ i-w \\ j-w \end{bmatrix},$$

$w = p + q - 1$.



Slika 8.29: Graf poteka signalov za množenje pasovnih matrik ($p = q = 4$)

Graf poteka signalov, ki ga dobimo s to preslikavo, je pravokotno polje velikosti $p \times q$ (slika 8.29).

Če uporabimo časovna pravila, dobimo sistolično polje na sliki 8.30, perioda razširjanja podatkov je $\alpha = \mathbf{s}^{\mathrm{T}}\mathbf{p} = 3$.

8.5.8 Reševanje sistemov linearnih enačb

Pomemben matrični problem, s katerim se srečujemo pri obdelavi signalov, je reševanje sistemov linearnih enačb. Običajo so to sistemi n linearnih enačb z n neznankami.

V matričnem zapisu opišemo problem z iskanjem vektorja $\mathbf{x}$, ki ustreza matrični enačbi

$$\mathbf{A}\mathbf{x} = \mathbf{y},$$

za znana A in y. Ponuja se rešitev

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{y},$$

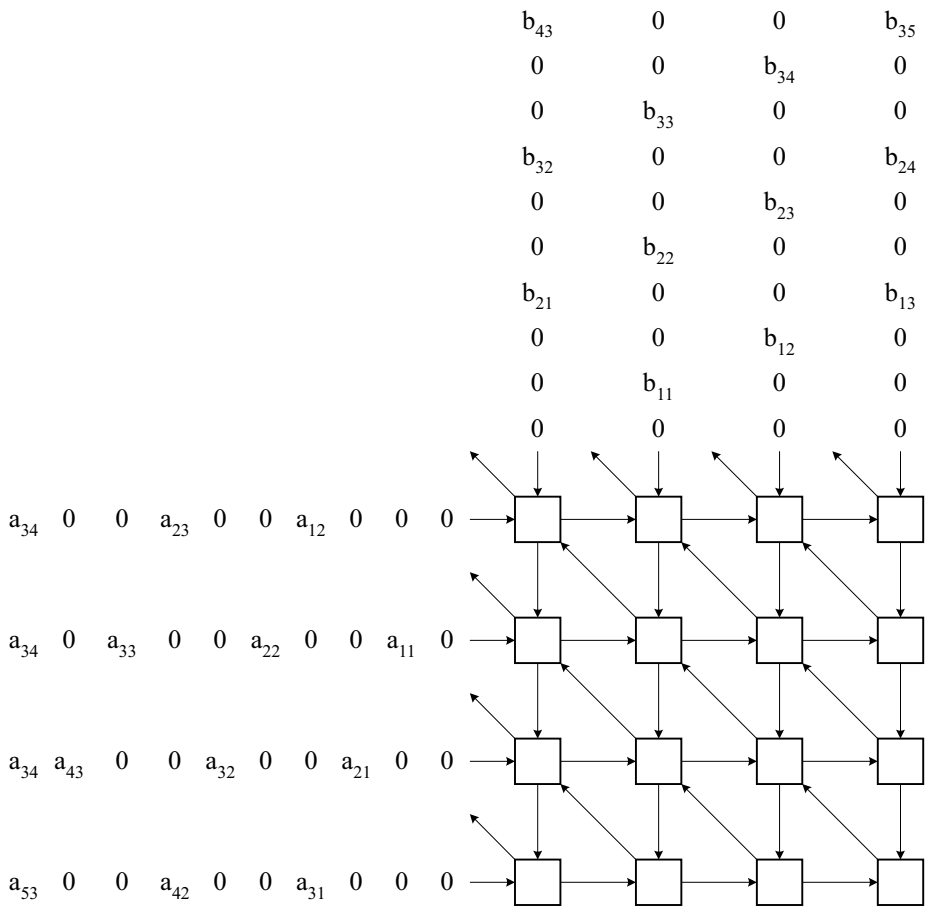
vendar je računanje inverzne matrike A v splošnem kompleksno in zaradi velikega števila računskih operacij neprimerno. Zato običajno sistem preoblikujemo v obliko

$$\mathbf{A}'\mathbf{x} = \mathbf{y}',$$

kjer je A' zgornje trikotna matrika. rešitve za x dobimo s pomočjo povratnih substitucij, na primer:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6 & 3 & 1 \end{bmatrix}$$

$$x_3 = 1$$
$$x_2 + 2x_3 = 3$$
$$x_1 + 2x_2 + 3x_3 = 6$$



Slika 8.30: Sistolično polje za množenje pasovnih matrik

$$x_3 = 1$$
$$x_2 = 1$$
$$x_1 = 1.$$

Eden od postopkov za preoblikovanje matrike $\mathbf{A}$ v zgornje trikotno obliko je t.i. QR dekompozicija. Matriko $\mathbf{A}$ razstavimo na produkt matrike z ortonormalnimi stolpci $\mathbf{Q}$ in zgornjetrikotne matrike $\mathbf{R}$, $\mathbf{A} = \mathbf{QR}$. Za QR dekompozicijo je na voljo nekaj postopkov; oglejmo si postopek z uporabo t.i. Givensove rotacije.