LOGIČNO NAČRTOVANJE RAČUNALNIŠKIH STRUKTUR IN SISTEMOV

N. Zimic, J. Virant

Fakulteta za računalništvo in informatiko v Ljubljani
Tržaška 25, 1000 Ljubljana, Slovenija

Maj 1998

Kazalo

UVODNA BESEDA

Pričujoče delo predstavlja učbenik za logično načrtovanje (snovanje) računalniških struktur in sistemov. Namenjen je predvsem študentom in inženirjem računalništva in informatike. Ker pa so vsak dan bližji, in tudi do določene mere že, integrirani sistemi področij računalništva, informatike, komunikacij, telekomunikacij in mikroelektronike, je samo po sebi razumljivo, da je učbenik primeren tudi za študente in inženirje ostalih omenjenih področij dela. V zadnjih desetih letih ni izšel noben omembe vreden učbenik za načrtovanje računalniških struktur in sistemov, ki bi bil napisan v domačem jeziku. Ravno v tem obdobju pa je prišlo do velikih, relativno naglih, sprememb in novih smeri načrovanja računalniških struktur in sistemov. S tem domačim učbenikom želimo premostiti vidno vrzel in zastarelost znanj, zapisanih v domačem jeziku.

Delo je razdeljeno v štiri dele: Fizična in logična modularnost računalniških struktur in sistemov, Programirnost gradnikov v računalniških strukturah in sistemih, Upoštevanje časa in distance v računalnikih strukturah in sistemih in Postopki načrtovanja računalniških struktur in sistemov. Prvi del obravnava postopke in metode, na osnovi katerih lahko postavljamo računalniško strukturo na računalniško kartico, ki jo vstavimo v računalniški sistem. Vlogo računalniških kartic pa v vedno večji meri zmanjšujejo in zamenjujejo programirni VLSI-gradniki (PAL, GAL, PLD, Xilinx-, Altera- in drugi ASIC-gradniki). Prav zato je v drugem delu učbenika kar precej prostora namenjenega programirnosti takšnih gradnikov. Napravljen je prehod iz klasičnih tiskanin na programirna povezovalna in seveda tudi logična polja v gradnikih.

Značilnost razvoja zadnjih deset let je hitro povečevanje hitrosti delovanja gradnikov struktur in kompletnih računalniških sistemov. Tako na primer že v navadnem PC dosegamo hitrosti 200, 300 MHz, čeprav še ni daleč čas, ko je bila v PC desekrat nižja hitrost delovanja. Naglo krajšanje osnovne urne periode v računalniških strukturah in sistemih je v načrto-

vanje pritegnilo dodatne postopke in metode dela. Današnje kvalitetno načrtovanje na osnovi programirnih gradnikov že zahteva preučevanje asinhronskih vplivov dolžine povezave, pozicije gradnika, čipa, vrat na neki površini. Če ne znamo pravilno (logično) upoštevati teh vplivov, pride lahko do drugačnega logičnega delovanja, kot si ga želimo. Tem problemom je namenjen tretji del učbenika, v katerem sta preučevana prav distanca in čas v logični strukturi in med takšnimi strukturami.

V zadnjem delu učbenika sta vzeta v presojo avtomat in diagram poteka. Podani so postopki, ki omogočajo, da dokončno oblikovani uporabniški ali sistemski diagram poteka realiziramo kot avtomat, grajen s programirnimi gradniki. Ker se računalniški sistemi vedno bolj porazdeljujejo na osnovi komunikacijskih protokolov, postajajo nepregledni tudi na področju logičnega načrtovanja. Da bi povečali preglednost porazdeljenih računalniških sistemov, smo v učbenik vpeljali jezika SDL in VHDL, ki igrata v tem času osrednjo vlogo pri opisovanju, specifikaciji in načrtovanju računalniških in seveda tudi komunikacijskih sistemov. Preko SDL prihajajo v sistem bolj komunikacijski in preko VHDL bolj računalniški sestavi, čeprav kažeta oba na splošno univerzalnost.

Na koncu dela je Dodatek, v katerem so podani programi, ki dodatno in natančnejše opredeljujejo nekatere postopke, opisane v delu. Programi sodijo v okvir okolja MATLAB V5, v katerem je uporabljen optimizacijski paket Optimization Toolbox. Na koncu Dodatka so vprašanja in odgovori, ki omogočajo, da lahko bralec tudi na ta način aktivno razmišlja o podani snovi.

Učbenik daje znanja za logično načrtovanje računalniških struktur in sistemov, kar pa še niso globalna znanja o organizacijah in arhitekturah računalniških sistemov. Vsekakor pa so znanja v tem učbeniku dobra osnova, na katero se lahko postavljajo globlja spoznavanja računalniških organizacij in arhitektur. V kolikor pride bralec do občutka, da je slog opisovanja v tem delu preveč jedrnat pri nekaterih osnovnih razlagah, mu priporočamo, da si takšne razlage ogleda v [1].

Zahvala

Na tem mestu se zahvaljujeva red. prof.dr. Tomažu Slivniku, dipl. mat. in red. prof.dr. Andreju Dobnikarju, dipl. ing. za oceno dela in pomembne nasvete za izboljšavo le-tega, ki sva jih pri končni ureditvi dela že upoštevala. Ministrstvu za znanost in tehnologijo Slovenije gre zahvala za sofinanciranje, ki je omogočilo izdajo dela. Založba Didakta, Radovljica že vrsto let izdaja med drugim tudi temeljnejša dela s področja računalništva, čeprav ta dela niso tako tržno zanimiva kot dnevna računalniška literatura, katere je polno v naših založbah. Ker je omenjena temeljnejša strokovna literatura nujno potrebna tudi v našem jeziku, se za gesto založbe želiva zahvaliti tudi v imenu drugih avtorjev.

Ljubljana, maj 1998.

N. Zimic, J. Virant

Del I

FIZIČNA IN LOGIČNA MODULARNOST

Poglavje 1

FAZE IN CIKLI NAČRTOVANJA RAČUNALNIŠKIH STRUKTUR IN SISTEMOV

1.1 UVOD

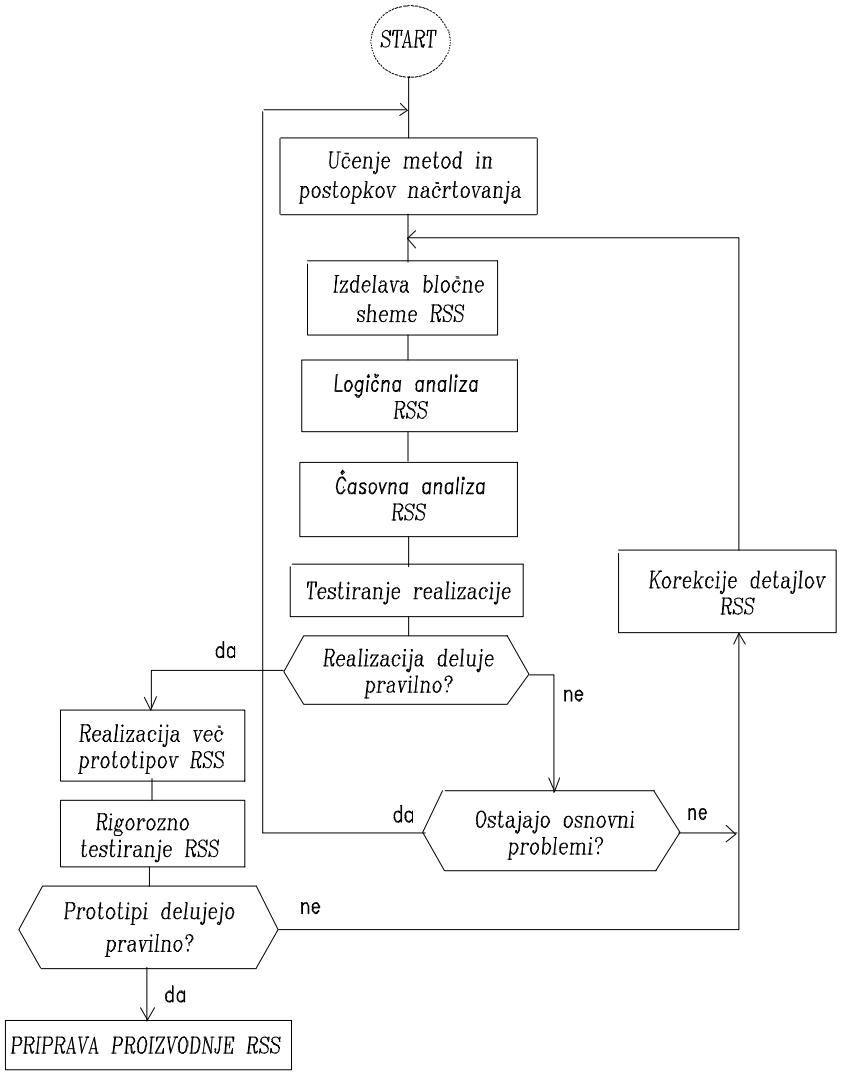
Računalniški sistemi, kar vsi po vrsti, opravljajo le tri različne kategorije funkcij: odločanje, pomnjenje in prenašanje informacij. Odločanje se izrazitejše nahaja v procesorjih in aritmetsko-logičnih enotah, pomnjenje v procesorjih in najrazličnejših pomnilnikih, ter prenašanje informacij v računalniških podsistemih, sistemih in mrežah. Sicer pa lahko katerokoli od omenjenih funkcij srečamo v kateremkoli računalniškem sistemu. Posebnost računalniških funkcij je tudi v tem, da so lahko porazdeljene. Tako lahko izvajamo odločanje med procesom pomnjenja, odločanje izvajamo med prenašanjem informacij in podobno. V tem delu bomo predsvem spoznavali metode načrtovanja (snovanja) računalniških struktur in sistemov (RSS) tako, da bodo ti sposobni opravljati omenjene funkcije.

RSS lahko svoja opravila izvajajo aparaturno, programsko ali mešano (firmware). Katerokoli aplikacijo ali sistemski postopek lahko zasnujemo na osnovi vsakega od omenjenih načinov, vedno pa obstajajo razlogi, zakaj pristopimo ravno na osnovi enega od treh načinov.

Po letu 1980 se je začela s presenetljivo naglico množično uveljavljati mikroelektronika s svojimi MSI- (medium), LSI- (large), VLSI- (very large) in tudi ULSI- (ultra large scale integration) čipi v smeri mikroprocesorjev in porazdeljenega procesiranja. Začeli so se postavljati, in seveda tudi uporabljati, programirni načrtovalni postopki aparaturne (strojne) računalniške opreme. Ta je postajala vedno bolj prijazna do računalniškega načrtovalca, tako, da je danes možno srečati aparaturna načrtovanja, ki so prijaznejša in hitrejša od čistih programskih.

Novejša tehnologija usmerja računalniške postopke, ki se pri procesiranju velikokrat ponavljajo oziroma uporabljajo (amortizacija postopkov), v čipne izvedbe. Takšne izvedbe so povsem primerne tedaj, ko imamo probleme z realnimi časi izvajanja postopkov. Že zadnje je dovolj, da postajata mikroelektronik in logični načrtovalec RSS vedno bolj pomembna in iskana strokovnjaka. Za sistemskimi postopki (operacijske rutine, prevajalniki, komunikacije, grafika itd.) je na vidiku, da se tudi velike, pomembne aplikacije lahko postavijo na en ali nekaj čipov. Zadnje lahko zelo zniža stroške izvajanja (uporabljanja), poveča zanesljivost in pomanjša volumen računalniškega oziroma informacijskega sistema in prostora.

Že v klasičnem računalniškem obdobju (1940 do 1970) so prišli do diagrama poteka načrtovanja RSS, ki ga podajamo na sliki 1.1. V diagramu vidimo dve zanki, ki zahtevata večkratno delo v okviru postavljanja in popravljanja logičnih shem (načrtov), nove oziroma popravljene prototipe, analizo RSS po logiki in času, iskanje logičnih napak, odpovedi komponent itd. Vstopna točka v diagramu poteka na sliki 1.1 se nanaša na poznavanje postopkov načrtovanja, izhodna točka pa predvideva, da je (dokumentiran) načrtovalčev izdelek primeren za pripravo proizvodnje RSS. Sodobno računalništvo je značilno po tem, da se klasični model na sliki 1.1 v veliki meri podpira z računalniškimi orodji, pri tem pa se daje velik poudarek na logično in časovno analizo ter testiranje (večkrat tudi 100 %). Diagram poteka načrtovanja lahko računalniško v celoti podpremo, vendar je načrtovanje RSS vedno večja stroškovna postavka. Ura razvoja na veliki delovni postaji, ki pokriva vse faze na sliki 1.1, je stroškovno zahtevna, saj mora razvoj pokriti velike stroške za nakup računalniškega orodja in delovne postaje ter plačilo načrtovalca. Drugače je v primeru, če imamo za delovno postajo kar PC in na njem manjša cenena računalniška orodja za načrtovanje RSS. Seveda pa lahko na PC načrtujemo le manjše RSS. Načrtovanje poprečne RSS je vendarle povezano s tolikšnimi stroški, da zahtevamo od načrtovalca velik delež znanj in, da, na osnovi tega, posega po delovni postaji samo takrat, ko je to potrebno in se to tudi izplača. Zelo ve-



Slika 1.1: Osnovne faze in cikli načrtovanja in razvoja RSS.

likokrat imamo centralno računalniško orodje za načrtovanje RSS, katerega lahko hkrati uporablja več deset načrtovalcev vsak s svojega zornega kota načrtovanja. V takšnem primeru se mora načrtovalec zavedati, da je on le del večje načrtovalne celote, v kateri mora veljati disciplina, red in toleranca.

Poskušali bomo obdelati znanja, ki so potrebna pri sodobnem načrtovanju RSS. Pri načrtovanju RSS se nam računalnik pojavlja dvakrat: prvič kot orodje in drugič kot objekt načrtovanja. Te posebnosti nimamo pri nobenem drugem načrtovanju z računalnikom (na primer CAD (computer aided design) v strojništvu, CAP (computer aided production) izdelkov, ACAD (Automatic computer aided design) itd. v vsej tehniki. Prav zato je snovanje in načrtovanje RSS še posebej zanimivo.

1.2 PROGRAMIRNOST RSS

Če je RSS majhna in industrijsko nezahtevna, jo lahko načrtujemo povsem neubrano. To pomeni, da na kateremkoli mestu v logični shemi RSS lahko postavimo katerikoli element. Važno je le, da izbira zadošča logični funkciji, na katero se nanaša shema. Če mora načrtovalec postaviti na primer datajl $x_1 \vee x_2$, lahko to stori z $\bar{x}_1 \rightarrow x_2$ (z implikacijo in negacijo), $\overline{x_1 \downarrow x_2}$ (s Pierceovo povezavo in negacijo) itd. V realizaciji z IC-vezji se s temi primeri spreminja tudi tip čipov, ki jih načrtovalec lahko postavlja kamorkoli na kartico. Omenjeno neubrano načrtovanje torej pomeni, da si načrtovalec na vsakem mestu sheme lahko privošči, kar mu pride na misel oziroma pod roke. Logične in druge elemente si lahko izmišljuje, seveda glede na svoje izkušnje in poznavanje načrtovanja.

Ne smemo si predstavljati, da je možnost neubranega načrtovanja stvar preteklega računalništva. Namreč, mikroelektronika ponuja vsak dan več programirnih čipov in pri programiranju le-teh ima načrtovalec veliko prostostno stopnjo načrtovanja. Neubranost je tako lahko zelo velika in če je ta strokovno optimalna, je tudi zaželena. Omenjena strokovna optimalnost je možna le na osnovi novih načrtovalskih znanj, ki se porojevajo z novimi družinami velikih programirnih čipov.

Sodobne RSS kaj hitro preraščajo v velike sisteme, v katerih ni vedno vse tako, kot si želi načrtovalec. Elektronska in informacijska tehnologija ponujata na vsakem koraku vrsto modulov, podsestavov, VLSI-vezij, mikroprocesorjev itd., ki so vnaprej postavljeni in morda le 90 % ustrezajo tistemu, kar želi načrtovalec pri svojem delu. V takšnem primeru privzame omenjeni sestav in ga vgradi v RSS, ker je tak sestav ponavadi mnogo cenejši od pri-

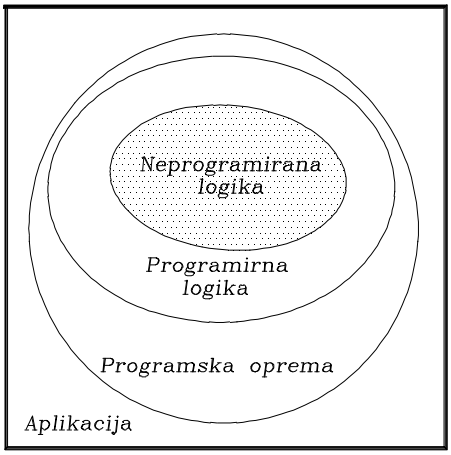
bližno enakega, katerega bi zasnoval sam. Če pa vgradi sestav, ki zadošča le 90 %, se mora največkrat krepko potruditi (znanje, inovacija, iznajdljivost), da postavi še manjkajočih 10 % sestava. V takšnem primeru govorimo o strukturalnem pristopu k načrtovanju RSS. Zlasti je pomembno, da zna načrtovalec prilagojevati VLSI- in ULSI-čipe za postavljene logične sheme, ki so v smislu realizacije še neopredeljene. Projekt je potrebno vedno tako pripraviti, da je uporabnost trenutno dosegljivih čipov velika.

S pojavom bralno-pomnilnih struktur se je pričel uveljavljati že omenjeni programirni pristop k RSS. Da bi načrtovalcem prepustili čim večjo prostostno stopnjo načrtovanja, so proizvajalci čipov začeli proizvajati programirne čipe. Ti na primer omogočajo, da zgoraj omenjenih 10 % načrtovalec nadoknadi s programirnim pristopom. Seveda pa so cilji proizvajalcev čipov v tem času še mnogo višji. Poskušajo dati na tržišče programirne čipe, na kateri je možno programirati skoraj vse: odločitvena vezja, pomnilna vezja in tudi povezave med takšnimi vezji. Tako postajajo čipi vedno bolj univerzalni gradniki, v katere načrtovalec "vdahne dušo" načrtovane strukture oziroma sistema. Tako z istim čipom (ali nekaj čipi) lahko enkrat programira ALU enoto, drugič kontrolni avtomat, tretjič vmesnik in podobno. Orodje za načrtovanje je v vseh teh primerih enako.

Čim več je programirnosti na čipih, več časa načrtovalec prebije ob tipkovnici delovne postaje. Pravzaprav je v tem času že možna aparaturna oprema, ki je postavljena povsem programsko (handless CAD and CAP). Razlika med programerjem na računalniku in načrtovalcem RSS je v tem, da programer izdela na primer programski paket, načrtovalec pa čipno strukturo (sistem). Zadnja struktura izvaja računalniške funkcije na aparaturni način, medtem ko programerjev paket izvaja funkcije na programski način - morda na aparaturni opremi, ki jo je na programirni način postavil omenjeni načrtovalec RSS.

Mikroračunalniško načrtovanje predvideva, da so omenjeni univerzalni programirni čipi v večji meri kar mikroprocesorji. V določeni RSS lahko srečamo več mikroprocesorjev naenkrat, seveda okoli njih pa še tudi ostalo programirno in neprogramirno logiko. V takšnem primeru lahko postavljamo RSS, ki ima porazdeljeno moč odločanja, pomnjenja, kontrole in podobno.

Računalniški sistem lahko z vidika načrtovanja opazujemo v slojih, kot to vidimo na sliki 1.2. V sredini je, kot jedro, negibka aparaturna oprema (pravi hardware), to obkroža gibka, to je programirna aparaturna oprema. Programirno logiko obkroža programska oprema in to, aplikacije, ki potrebujejo računalnik. Delitev neprogramirna aparaturna oprema-programirna



Slika 1.2: Slojnost glede na programirnost v RSS.

aparaturna oprema-programska oprema je zelo pomembna faza načrtovanja RSS. Od te faze v veliki meri zavisijo performance RSS na eni strani in stroški RSS na drugi strani. V posameznih delih lahko srečamo sestave, kot na primer:

- ☐ *Aparaturni del:* RAM, ALU, ROM, VODILA ...,
- ☐ *Programirni del:* PROM, REPROM, PAL, GAL, ALTERA, PLD, DSP, XILINX, QuickLogic QL, 4-, 8-, 16- in 32-bitni mikroprocesorji (INTEL, MOTOROLA), ASIC ...,
- ☐ *Programski del:* DOS, PREVAJALNIKI, KNJIŽNICE, BAZE PODATKOV ...

  itd.

Slika 1.1 je že dolgo časa osnovno vodilo pri načrtovanju RSS. Ta se s časom v principu ne menja, menja pa se zelo tehnika in orodje za načrtovanje. Vedno več je računalniških orodij za načrtovanje RSS, v katere je vgrajeno tudi vedno več potrebnih znanj. Ta znanja današnji načrtovalec niti ne obvlada več, čeprav je prav z orodji sposoben dobro načrtovati. Dobra stran načrtovalskih orodij je, ker

- ▷ zmanjšajo čas neposrednega načrtovanja,
- ▷ zmanjšajo čas, ki je potreben za izdelavo dokumentacije,
- ▷ se ta orodja lahko nadaljujejo v orodja za proizvodnjo RSS, po drugi strani pa lahko izhajajo iz specifikacijskih orodij RSS,

- ▷ zmanjšajo število napak v dokumentaciji,
- ▷ omogočajo sprotne simulacije in testiranje sheme,
- ▷ zmorejo avtomatično programirnost čipov,
- ▷ razbremenjuje preobremenjenega načrtovalca

  itd.

Sodobna orodja oziroma jeziki za specifikacijo in načrtovanje programirne aparaturne opreme so:

- ◦ HDL (Hardware description language),
- ◦ Verilog HDL,
- ◦ VHDL (HDL za VHSIC (Very high-speed IC)),
- ◦ CUPL (Hardware design language),
- ◦ ABEL(Advanced Boolean expressions language, -PAL, -HDL, -FPGA),
- ◦ PALASM (PAL asembler, -TOABEL),
- ◦ APLUS (Altera PLUS),
- ◦ Electronics Workbench (The elektronics lab in computer)

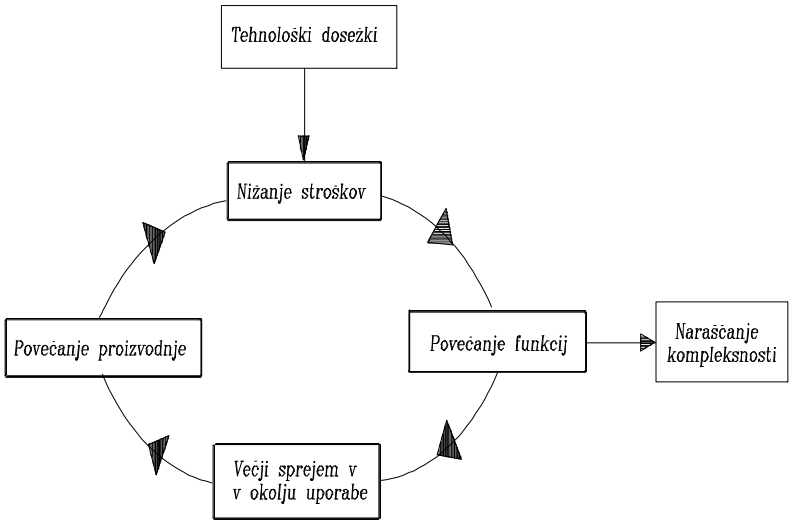
  in druga.

Ker se dajo aparaturni gradniki programirati in programe postavljati tudi aparaturno, lahko rečemo, da med aparaturno in programsko opremo ni bilo še nikoli toliko mehkega medsebojnega prehajanja kot v tem času. V računalniški industriji dobiva smisel tako imenovani "hardware/software co-design" [24], ki ugodno vpliva v vseh pogledih. Zlasti je pomembno, da se moramo prav takšnemu načrtovanju zahvaliti, da imajo mikroprocesorji, mikroperiferija, mikrokontrolerji in procesorji signalov skupaj 34 % tržno rast v obdobju 1994 - 1998 (podatek po *Dataquest*). Samo tržnost mikroprocesorjev presega 25 milijard US $ letno.

Primerna delitev RSS na programski in aparaturni del, z zelo izraženo prekrivno programiranostjo med obema, močno vpliva na dokončne performanse RSS, na nabor instrukcij RSS, porabo energije za delovanje RSS, velikost RSS, ceno RSS itd. To so tako pomembne postavke, da mora vsak načrtovalec dobro spoznati strukturnost, organizacijo in arhitekturo RSS. Ta učbenik uči le osnovni pristop k programirnosti RSS z vidika logičnih in fizičnih shem, za ostalo pa so nam na voljo drugi učbeniki. Načrtovanje bomo delili v dva dela:

- ☐ Fizično načrtovanje in
- ☐ Logično načrtovanje.

V prvem primeru nas zanima predvsem matematizacija fizične modularnosti RSS in v drugem, postavljanje logičnih shem, modulov in realizacij RSS. Eno in drugo sodi med osnovna znanja, ki jih potrebujemo predhodno,



Slika 1.3: Razvojno-proizvodni cikel RSS.

preden se spustimo globlje v organizacijo in arhitekturo RSS. Seveda je pri tem poglabljanju, poleg omenjenega znanja, potrebno še znanje o sami organizaciji in arhitekturi RSS.

1.3 SOVISTNOST NAČRTOVANJA IN PROIZVODNOSTI RSS

Pri načrtovanju RSS se moramo zavedati povratnega vpliva s strani njene proizvodnosti. Cikel proizvodnosti prikazuje slika 1.3. Za načrtovanje RSS so pomembni tehnološki dosežki, ki vplivajo na nižanje stroškov proizvodnje. Znižanje teh stroškov ima lahko za posledico povečanje funkcij (lahko performans RSS), to povečanje funkcij pa lahko močno povečuje kompleksnost RSS in stroške načrtovanja. Zaradi velike sinergije med načrtovanjem, razvojem in proizvodnjo RSS se računalniška orodja vseh treh področij dela prelivajo oziroma prekrivajo. Najbrž pa bo čas prinesel enotno orodje, ki bo zmoglo postaviti vse postopke, ki bodo ležali med osnovno specifikacijo podatkov in končnim izdelkom RSS.

Stroški v US $	FPGA	MGA	CBIC
Stalni proizvodni stroški	21800	86000	146000
Variabilni stroški na čip	39	10	8

Tabela 1.1: Primerjava stroškov različnih ASIC-tehnik.

1.4 VPLIVNOST STROŠKOV NA NAČRTOVANJE

Pri naglem tehnološkem razvoju zlasti IC-vezij in pomnilnih medijev se je načrtovalcu RSS včasih kar težko znajti, ko gre za stroškovnost načrtovanja in proizvodnjo RSS. Razne tehnike so stroškovno gledano lahko zelo različne med seboj. Za primer si poglejmo samo različne ASIC-tehnike (application-specific integrated circuits), ki so v tem času na voljo načrtovalcu. Kot bomo spoznali kasneje, imamo v principu tri tehnike [15]:

- CBIC (cell-based ASIC), osnovna značilnost: na čipu so vnaprej pripravljene standardne celice (AND-vrata, OR-vrata, MX, pomnilne celice itd),
- MGA (masked gate array), osnovna značilnost: maskirane porazdelitve vrat (GA) in
- FPGA (field-programmable gate array), osnovna značilnost: kompleksni PLD (programmable logic device).

FPGA-čip je pri načrtovanju sicer privlačen, vendar je hrati tudi najdražji. Tako je 20K GA MGA pri 0.5-mikronski tehniki 2 - 4 $ (pri nakupu 10000 kosov velja 0.01 -0.02 centov/vrata), medtem ko je GA enakega reda velja približno 20 $. MGA in CBIC je v poprečju 2 - 5 krat cenejši od FPGA. Pomembna je količina potrebnih čipov. Pri količini 10 - 1000 kosov je cenejša FPGA-tehnika, medtem ko sta pri količini 5000 - 100000 kosov cenejši MGA- in CBIC-tehnika. Nad količino 12000 kosov je CBIC-tehnika cenajša tudi od MGA. Za GA z 10K vrati velja primerjava v tabeli 1.1. Ker stroški v drugi vrstici tabele naraščajo, v tretji pa padajo imamo v opazovanem področju stroškov nekje sečišče, ki odločno posega v odločanje pri izbiri tehnike in količine potrbnih primerkov.

V letih 1986 do 1996 je IC-tehnologija prešla iz 2-mikronske na 0.6-mikronsko tehniko. Pri tem je performančni indeks cena/vrata padel iz 1

centa/vrata na 0.01 centa/vrata. Če si samo ta performančni indeks predstavljamo v okviru postavke Tehnološki dosežki na sliki 1.3, vidimo, kako učinkovite so lahko spremembe v razvojno-proizvodnem ciklu na omenjeni sliki.

Poglavje 2

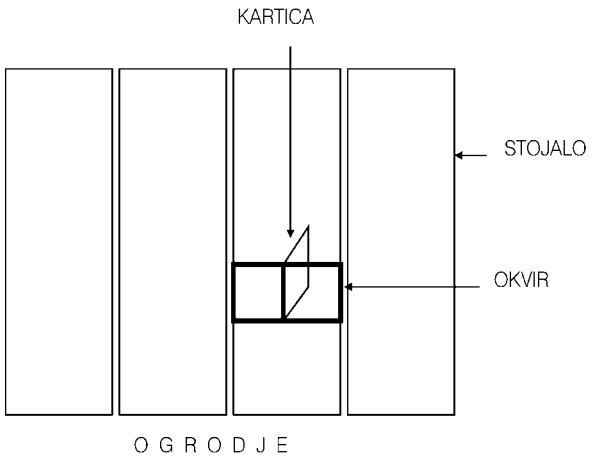
FIZIČNO NAČRTOVANJE RSS

2.1 FAZE DELA PRI POSTAVLJANJU RAČUNALNIŠKE KARTICE

Preden se lotimo logičnega snovanja RSS, si oglejmo fizično okolje, v katero se nameša takšne strukture in sisteme. SSI-, MSI-, VLSI-, ULSI- in druge računalniške komponente nameščamo na kartice (plug-in-unit, card, PCB), te vstavljamo v okvirje, okvirje v stojala in stojala v ogrodja oziroma ohišja, kot to prikazuje slika 2.1. V našem primeru bomo opazovali predvsem kartico, saj je ta objekt največkrat obravnavan s strani načrtovalca. Kartica je največkrat tiskanina, na kateri so čipi in druge komponente, ki zaokrožajo določeno logično ali funkcionalno celoto. Pomembna je ugotovitev, da je kartica je najmanjši del sistema, ki je fizično hitro odtujiv, odstranljiv in zamenljiv.

Pri neposrednih postavitvah komponent na kartice srečamo vsaj naslednje faze dela:

1. razdelitev velikega preklopnega vezja na smiselna podvezja (module) (partitioning of circuit),
2. izbira in označitev IC-komponent, ki realizirajo podvezja iz točke 1 (IC-selection and assignment),
3. izbira in označitev množice IC-komponent, ki sodijo na eno kartico (card specification and assignment),
4. namestitev IC-komponent na kartico (IC-placement), označitev priključkov na kartici (pin assignment),



Slika 2.1: Konstrukcijski sestav večje RSS.

5. povezovanje podvezij (modulov) na kartici (card routing),
6. namestitev kartic v okvir (card placement) in
7. povezovanje med karticami v okvirju (backplane routing).

Problemi, ki se pojavijo pri omenjenih fazah dela, velikokrat niso enostavni, če jih gledamo z vidika kombinatorike. Zato načrtovalci težijo k intuitivnim, rutinskim in drugačnim rešitvam, saj jih kratek čas načrtovanja, razvoja in realizacije potiska k prav takšnim rešitvam. Vzemimo na primer, da je na kartici n pozicij in da moramo na to kartico namestite m IC-vezij. Če je $n \geq m$, obstaja

$$N = m! \binom{n}{m} \tag{2.1}$$

možnih namestitev, vsako takšno namestitev pa spremlja veliko število tehnoloških ugodnosti in neugodnosti, o katerih mora razmišljati načrtovalec kartice. V primeru ko je $n < m$, ne moremo vseh modulov položiti na kartico, vendar je včasih možno problem rešiti tako, da se nekateri moduli polagajo en na drugega (na primer zasilne rešitve pri modulih elektronskih pomnilnikov).

Zgled 1 *:*

Ocenimo število možnih namestitev, če je $n = m = 10$.

$$N = \frac{10! \cdot 10!}{(10-10)! \cdot 10!} = 3628800.$$

Vidimo, da povsem normalen velikostni red dimenzij n in m da toliko možnih namestitev, da načrtovalec ne more ubrati na primer analitično pot optimizacije nameščanja.

Ker so problemi postavljanja objektov na kartico lahko obsežni, obdelati je potrebno obilo kombinacij, permutacij in podobnih variacijskih značilnosti, velikokrat posegamo po hevrističnih metodah in postopkih reševanja. Uveljavljajo se zato tudi računalniška orodja z vgrajenim izkustvenim znanjem (baze znanj, expertni sistemi), ki pa so navadno tako dragi, da so srednjim in manjšim načrtovalskim laboratorijem nedostopni.

2.2 SPREMLJAJOČI STANDARDI

Do določene realizacije RSS lahko pridemo na neskončno mnogo načinov. Da lažje in boljše načrtujemo, razvijamo in proizvajamo RSS, so na voljo razni standardi, predpisi, navodila za izvajanje načrtovalnih in tehnoloških postopkov itd. Vsak proizvajalec RSS dobro ve, da delo pri načrtovanju, razvoju in proizvodnji poteka dovolj strokovno le, če se drži potrebnih napotil in standardov. Tako imamo:

- standarde za logično načrtovanje,
- elektriške standarde,
- mehanske standarde,
- predpise,
- priporočila,

itd.

Ti so lahko izdelani na svetovnem, evropskem (mednarodnem), nacionalnem ali firmnem nivoju. V razvitem svetu naj bi firmni standardi upoštevali čim več nacionalnih izhodišč, ta zopet čim več mednarodnih izhodišč itd. Pa vendar ob vsem tem ostane še vedno veliko internih izhodišč, ki so značilna le za opazovano firmo. V tem delu se nimamo namena ukvarjati s standardi in tehnologijami posameznih proizvajalcev, ker ustrezna znanja ne sodijo v okvir splošnega študija. Firmna znanja so dopolnilna znanja, za katere poskrbi sama firma. Ta znanja firma daje inženirjem, ki so že pridobili splošna znanja. Za standarde na splošno pa lahko rečemo, da imajo v

tem času izreden pomen in da ta pomen vedno bolj narašča. Če želi računalnikar na primer načrtati in razviti računalniško-komunikacijsko napravo ali sistem (modem, ISDN-vmesnik, računalniško mrežo itd.) se znajde že prvi dan pred nekaj tisoč stranmi standardov in priporočil, ki ga spremljajo vse do zadnjega trenutka načrtovanja in razvoja RSS. Mednarodne institucije, ki se ukvarjajo s standardi, ki vsaj v določeni meri zadevajo RSS, so: IEC (International Electrotechnical Comission, CEPT (European Conference of Postal and Communication Administration), CCITT (International Organization for Telegraph and Telephon Conssultative Committee), IEEE (Institute of EE Engineers), ISO - International Organization for Standardization, IEA (Electronic Industries Association), in še nekatere druge.

V našem izvajanju so predvsem pomembni standardi, ki se nanašajo na logične sheme in fizično postavljanje le-te. Predvsem se ne moremo izogniti standardom:

1. ANSI, American National Standards Institute (podajanje logičnih shem v ZDA),
2. DIN, Deutsche Industrienorm (podajanje logičnih shem po nemških standardih DIN 40 700) in
3. IEC, International Electrotechnical Commission (evropski standardi za podajanje logičnih shem IEC 617-12).

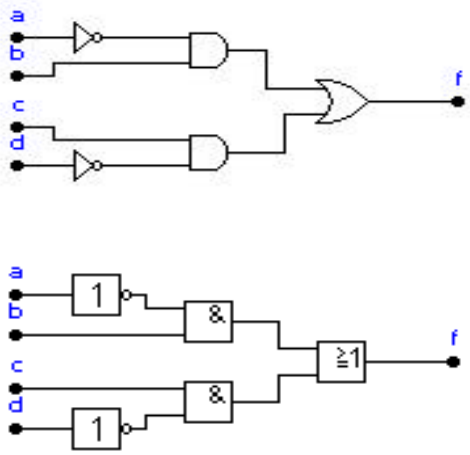
2.3 LOGIČNI IN MATEMATIČNI GRAF PREKLOPNEGA VEZJA

Logično vezje je v tem učbeniku takšno vezje, ki realizira določeno logično funkcijo oziroma logično shemo, s tem da je preklopna spremenljivka preprosta Booleova spremenljivka. Logične sheme lahko podajamo:

- ▷ povsem prosto, ne glede na kakršenkoli standard ali priporočilo,
- ▷ na osnovi firmnih standardov,
- ▷ na osnovi nacionalnih priporočil ali
- ▷ na osnovi mednarodnih priporočil.

V okviru omenjenih standardov oziroma priporočil si oglejmo preklopno funkcijo

$$f(a,b,c,d) = \bar{a}\,b \vee c\,\bar{d}. \tag{2.2}$$

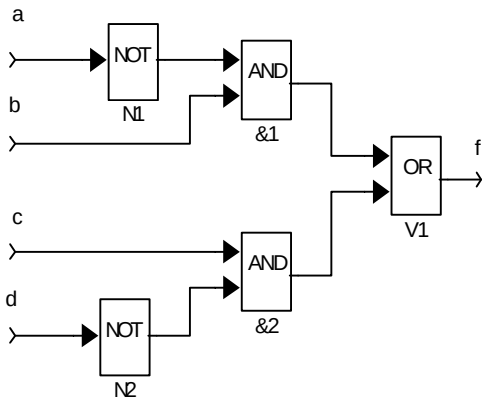


Slika 2.2: Logična shema po priporočilih ANSI (zgoraj) in po priporočilih IEC (spodaj).

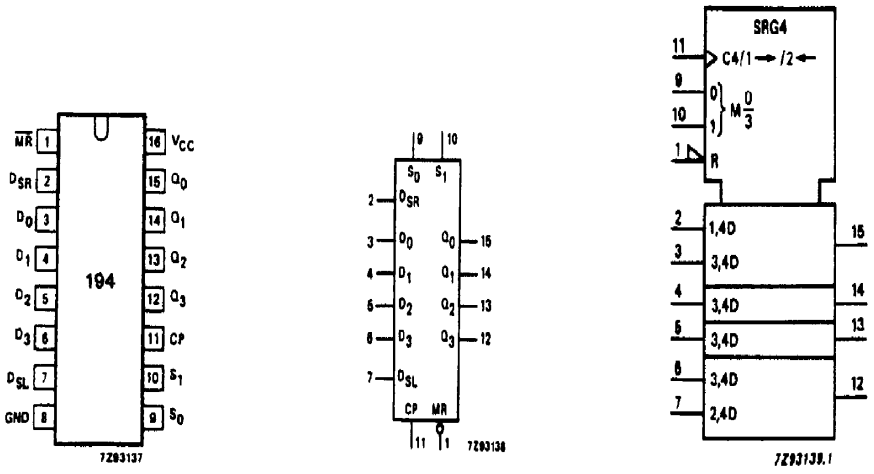
Zgornja predstavitev na sliki 2.2 se nanaša na ANSI in spodnja na priporočila IEC 617-12 [2],[22]. Vidimo, da je spodnji primer primernejši za risanje simbolov, tudi z vidika računalniške grafike. Slika 2.3 prikazuje opazovano logično shemo na osnovi logičnih simbolov orodja MATLAB&SIMULINK, ki med drugim lahko simulira tudi logične sheme.

Vsak evropski računalniški inženir naj bi obvladal logične sheme, če so te narisane na osnovi priporočil IEC. Zato naj ne bo odveč, če si pogledamo ta standard vsaj na enem primeru. Vzemimo MSI-vezje 74HC/HCT194 na sliki 2.4. Ta čip vsebuje 4-bitni premikalni register v obe smeri, vendar konfiguracija priključkov čipa na levi strani omenjene slike ne podaja nobenih logičnih značilnosti takšnega registra. Tudi logični simbol v sredini slike ne pove skoraj nič o logičnem delovanju registra. IEC-simbol registra na desni strani slike 2.4 nakazuje posamezne logične celote in odvisnosti. Ob primernem izkustvu se da ugotoviti celotno logično delovanje čipa. Potrebno pa je vendarle reči, da za detaljno logično analizo potrebujemo logično shemo, ki je na sliki 2.5 podana na nivoju vrat.

Na sliki 2.4 je IEC simbol predstavljen z gornjim delom, ki je v spodnji del na nek način nataknjen. Zadnje pomeni, da je gornji del kontrolni del, ki generira kontrolne argumente, ki so skupni vsemu spodnjemu delu modela. Tako sta na priključkih 9 in 10 selekcijska vhoda S0, S1 z notranjo oznako



Slika 2.3: Logična shema na osnovi logičnih simbolov orodja MATLAB-&SIMULINK.



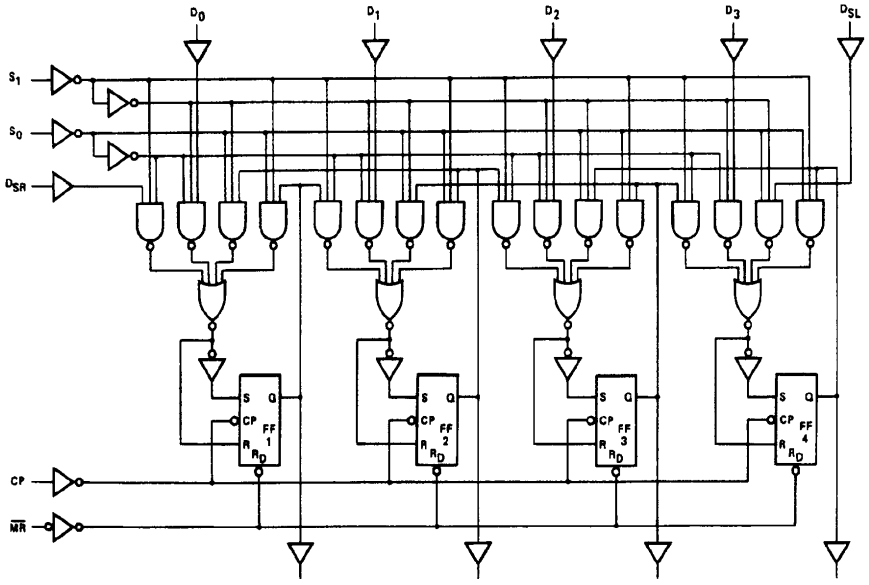
Slika 2.4: Premikalni register 74HC/HCT194, priključki (levo), logični simbol (v sredini), IEC-logični simbol (desno).

$$\left.\begin{matrix}0\\1\end{matrix}\right\} \mathsf{M}\tfrac{0}{3}.$$

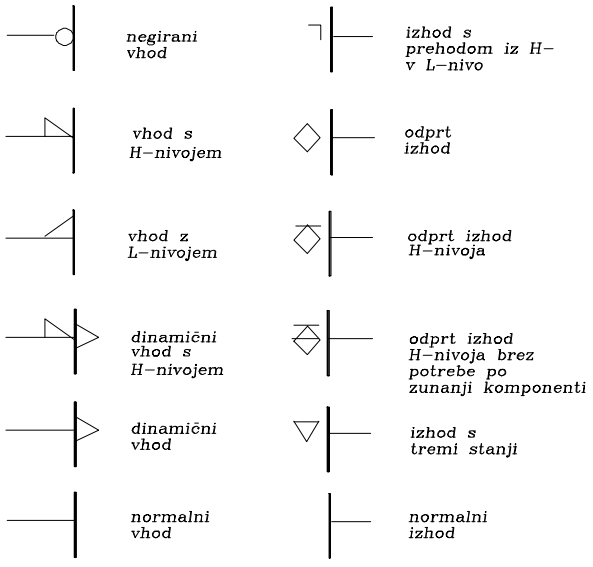
Zadnji zapis pomeni, da gre za delovne načine M0/3 = 0 (sprejeti impulzi nimajo izhodnega učinka), M0/3 = 1 (desni premik vsebine), M0/3 = 2 (levi premik vsebine) in M0/3 = 3 (paralelni vhodi D0, D1, D2, D3 (priključki 3, 4, 5, 6). V argumentni del zgoraj simbola spada še vhod R za brisanje vsebine registra (vpliv zahteva L-logični nivo) ter ura CP. Priključek ure 11 ima v argumentnem delu oznako puščice, kar pomeni, da gre za frontno (dinamično) vplivanje na vse 4 pomnilne celice. Puščici pomenita, da se ob urnem signalu vsebina premika levo ali desno. C4 pomeni, da imamo opraviti s kontrolnim vhodom za sekvenčno preklopno vezje. V spodnem delu modela so pomnilne celice z D oznakami. Oznaka 1,4D pomeni, da gre za desne serijske podatke, ki imajo dostop, ko je M0/3 = 1, (S0, S1) = (0, 1). Oznaka 2,4D pomeni, da gre za leve serijske podatke, ki so aktivni v primeru M0/3 = 2, (S0, S1) = (1, 0). Pri oznaki 3,4D imamo paralelni vnos podatkov, ko je M0/3 = 3, (S0, S1) = (1, 1). Priključek 2 tako predstavlja desni serijski vhod, priključek 7 levi serijski vhod in priključki 3 - 6 paralelni vhod štirih podatkov. Na desni strani IEC simbola so vedno izhodni priključki. V našem primeru imamo tako priključke 15, 14, 13, 12 za Q0, Q1, Q2, Q3. Na priključku 15 imamo levi serijski in na 12 desni serijski izhod. Izhodni priključki 15 - 12 pa predstavljajo tudi paralelni izhod, ko je M0/3 = 3, (S0, S1) = (1, 1). D v spodnjem delu IEC sheme pomeni, da gre za pomnilno celico z vhodom $D = S = \bar{R}$ (edge-triggered flip-flop).

Še nekaj podatkov, ki jih v primeru čipa 74HC/HCT194 ne zasledimo sicer pa so pogosti v IEC-shemah [22]. V kontrolnem, argumentnem delu poleg načina delovanja M lahko srečamo še način G, ki pomeni omogočenost (enable) za vhodne podatke (ta implicitno igra tudi vlogo argumenta konjunkcije). Omogočenost izhodnih podatkov predstavlja ime EN. R pomeni asinhronsko brisanje celic. Na sredini gornjega dela je ime enote. Imena so sicer poljubna, vendar se tudi ta lahko jemljejo z naslednjim priporočilom

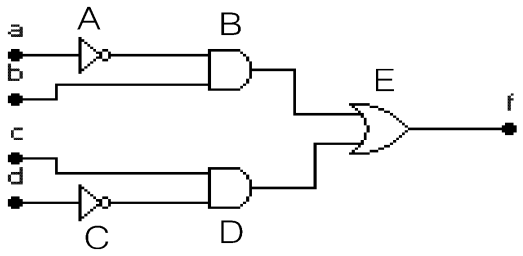
- SRGm, m-bitni premikalni register
- CTRm, m-bitni števec, cikel $2m$
- CTRDIVm, števec s ciklom m
- RAMmxn, pomnilnik z m besed dolžine n
- CAMmxn, asociativni pomnilnik kapacitete $m \times n$
- FIFOmxn, pomnilnik $m \times n$ kapacitet z značilnostjo najprej vhod - najprej izhod.



Slika 2.5: ANSI-logična shema za čip 74HC/HCT194.



Slika 2.6: Nekaj možnih vhodov in izhodov po priporočilih IEC.



Slika 2.7: Logični graf za funkcijo (2.2).

Nekaj osnovnih možnosti na vhodih in izhodih IEC-shem vidimo na sliki 2.6. H predstavlja logiko z gornjim nivojem in L logiko s spodnjim nivojem. Negacijo $\circ$ lahko srečamo tudi na izhodu, čeprav to na sliki ni prikazano. Normalna notranja puščica pomeni dinamični vhod. Zunanja polpuščica pomeni H- ali L-pristop, odvisno od tega, kako je obrnjena. Naj poudarimo, da odprt izhod lahko zahteva zunanjo komponento ali pa tudi ne. Razliko določa sredinska črta na znaku $\diamond$. Črta zgoraj in v sredini pomeni "pull-down" (na primer izhod na emitorju npn-tranzistorja), črta samo spodaj pomeni odprt kolektor npn-tranzistorja na izhodu. "pull-up" imamo v primeru črt v sredini in spodaj. Izhod je tedaj na primer na kolektorju npn-tranzistorja, itd.

Logični shemi (glej na primer sliko 2.5) lahko rečemo tudi logični graf. Ta poskuša opazovalcu predstaviti logično delovanje. Na osnovi obravnavanih logičnih shem lahko takoj formalno zapišemo logično funkcijo, ki jo uporabimo pri teoretičnem obravnavanju problema. Zadnja značilnost logične sheme pa je povsem nepomembna pri fizičnem postavljanju ustrezne RSS, to je pri realizaciji logične sheme. Pri fizični realizaciji je mnogo bolj pomembno, koliko operatorjev in modulov ima shema, koliko vhodov in izhodov ima, koliko priključkov imajo čipi, ali morata biti dva čipa oziroma modula blizu skupaj itd. Informacije v tem smislu dobi načrtovalec iz matematičnega grafa sheme in ne morda iz logičnega grafa, ki poskuša načrtovalcu "dočarati" samo logično delovanje sheme. Torej pomebno je razlikovati med pojmom matematični graf in pojmom logični graf, čeprav je, kot vemo, tudi logika pomembno matematično področje dela.

In kako postavimo matematični graf za določeno logično shemo? Za funkcijo (2.2) imamo logični graf na sliki 2.7. Vsakemu logičnemu operatorju v shemi dodelimo ime. Za 5 operatorjev v naši shemi izberemo na

primer imena A, B, C, D, E. Vsakemu imenu dodelimo vrh grafa, najboljše je kar z istim imenom. Če sta dva operatorja v logični shemi med seboj neposredno povezana povežemo ustrezna vrhova v grafu. Na ta način pridemo od slike 2.7 do slike 2.8. Graf na zadnji sliki ima vzdevek regularni (normalni) graf, ker vsaka poveza povezuje samo dva vrhova. Usmerjenost grafa bi lahko povedala, v kateri smeri teče informacija, vendar je ta podatek za fizično postavljanje logične sheme povsem nepomemben. Opazovani graf nas zadovolji, če ostane kar neusmerjen.

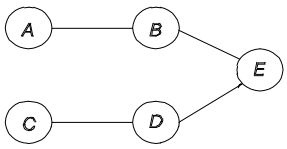
Pogost primer pri logičnih shemah je, da določena povezava povezuje več ali veliko operatorjev (argumentne povezave). V takšnem primeru posežemo po hipergrafih, v katerih je dovoljeno, da ena povezava povezuje več kot dva vrhova v grafu. Na sliki 2.9 vidimo logično shemo, kateri ustreza hipergraf na sliki 2.10. Kot vidimo, se v našem primeru povezava P_1 pojavlja šestkrat.

Matematični graf lahko ustreza logični shemi tudi tako, da ima dva tipa vrhov. Prvi na primer ustreza operatorjem in drugi povezavam sheme. Tak graf imenujemo bigraf. Za naš primer ga vidimo na sliki 2.11.

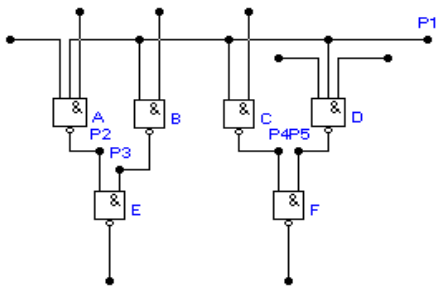
2.4 RAZDELITEV RSS NA MODULE

2.4.1 Uvod

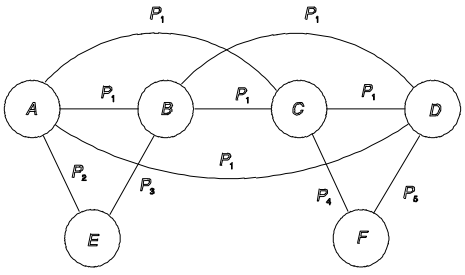
Vzemimo, da imamo na papirju ali grafičnem zaslonu dano veliko logično strukturo v slučajni, to je nemodularni logiki, in, da moramo pristopiti k modularno zastavljeni realizaciji te strukture. Strukturo moramo razdeliti na module in te povezati med seboj tako, da se ohranijo vse funkcijske in logične značilnosti strukture. Pri razdeljevanju strukture na module moramo paziti, da je medmodularnih povezav čim manj. Težimo torej k optimalni razdelitvi na module. Število povezav v notranjosti modula za nas ni pomembno, saj poskušamo pojem modul izenačiti s pojmom čipa. Če se nam zadnje posreči, se s problematiko notranjih povezav ne ukvarjamo. Te povezave prepustimo mikroelektroniku, kreatorju čipa, zato največkrat ne sodijo v domeno računalnikarjevega načrtovanja. Važno je le, da se zavedamo razlike v dolžini notranjih in zunanjih (medmodulskih) povezav. Medmodulska povezava je v primerjavi z modulsko vedno precej daljša in zato bolj problematična. Vpliv notranjih povezav zajamemo s časovno sliko delovanja ('timingom') čipa, ker je pri postavljanju RSS vedno pred nami v obliki dane dokumentacije. V tem podpoglavju si želimo ogledati medmodulske povezave, ki v veliki meri vplivajo na načrtovanje, v najnovešem



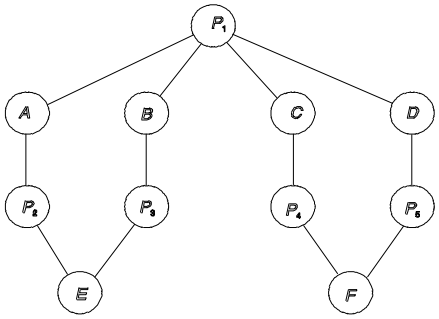
Slika 2.8: Regularni graf logične sheme na sliki 2.7.



Slika 2.9: Primer logične sheme z argumentno povezavo.



Slika 2.10: Hipergraf logične sheme na sliki 2.9.



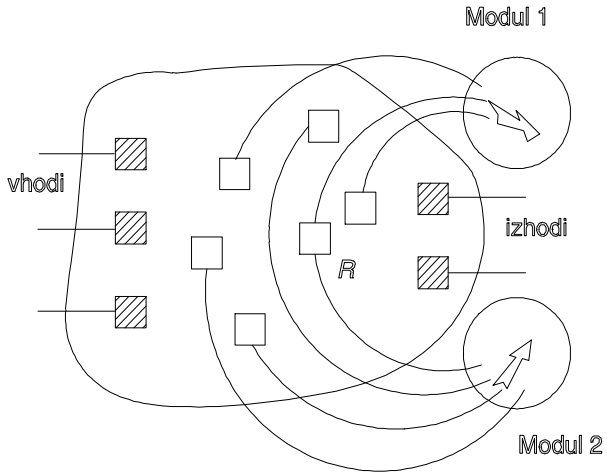
Slika 2.11: Bigraf za logično shemo na sliki 2.9.

času tudi na logično načrtovanje (PLD). Časovni vplivi predolgih povezav so naš problem, hitrost delovanja modula (čipa) pa problem elektronika. Pri načrtovanju poskušamo optimizirati tako število kot dolžino medmodulskih povezav. Če problem lahko rešimo z modulsko ali medmodulsko povezavo, vedno izberemo modulsko, saj ta vnaša vedno manj problemov od medmodulske vrstnice.

2.4.2 Replikacijska metoda

Metoda razdelitve na module se nanaša na module, v katerih se nahajajo logični operatorji oziroma skupine takšnih operatorjev (logični bloki). Modul ima svojo "kapaciteto", ki je podana s številom logičnih operatorjev v modulu in s številom priključkov na modul. Na modulu mora biti priključek za vsak zunanji vhod, za vsak zunanji izhod in za vsako ostalo medmodulsko povezavo. Posebnost metode je v tem, da dovoljuje v različnih modulih enake logične operatorje oziroma bloke. Ker je tem dodeljena tudi ista funkcionalnost, imamo opravka s kopijami - replikacijami; od tod tudi ime metode. Primer razdelitve logičnih operatorjev na dva modula prikazuje slika 2.12. Vidimo, da postavljamo operator R v oba modula z namenom, da pridemo do manjšega števila medmodulskih povezav. Vhodnih in izhodnih operatorjev (šrafirani moduli) ne razdeljujemo na module oziroma njihovo razdeljevanje na module je največkrat drugačno od notranjih operatorjev.

Metodo si oglejmo na primeru sheme, ki jo vidimo na 2.13. Preklopna struktura obsega operatorje z imeni 1 do 6 in povezave s $P1$ do $P9$. Naj en modul ne obsega več kot štiri operatorje in ne več kot štiri povezave. Za



Slika 2.12: Razdeljevanje logičnih operatorjev RSS na module.

vsak operator i lahko izpišemo množico povezav (i). Vsaka povezava v tej množici vodi do opazovanega operatorja i. Za sliko 2.13 imamo tako

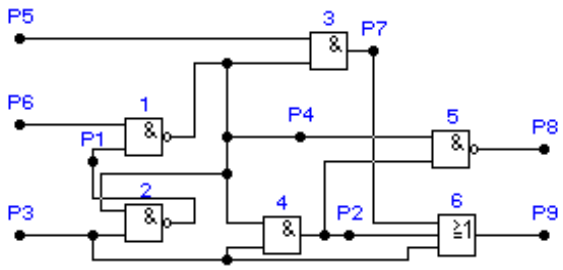
$$\begin{array}{l}(1) =< P1, P4, P6 > \\ (2) =< P1, P3, P4 > \\ (3) =< P4, P5, P7 > \\ (4) =< P2, P3, P4 > \\ (5) =< P2, P4, P8 > \\ (6) =< P2, P3, P7, P9 > .\end{array} \tag{2.3}$$

Vzemimo, da damo v modul z imenom M operatorje X, Y, Z in W. Množica vseh povezav (M) ni množica $(X) \cup (Y) \cup (Z) \cup (W)$, temveč množica $(X \cup Y \cup Z \cup W)$ z relacijo

$$(M) = (X \cup Y \cup Z \cup W) \subseteq (X) \cup (Y) \cup (Z) \cup (W). \tag{2.4}$$

Zadnjo relacijo omogočajo notranje povezave, ki jih v množici (M) ne podajamo.

Na sliki 2.13 lahko izberemo dva modula $M1$ in $M2$, ki vsebujeta pri danih pogojih naslednje operatorje



Slika 2.13: Označevanje logične sheme za razmeščanje elementov po modulih.

$$\begin{aligned} M1 &=< 1, 2, 4, 5 > \\ M2 &=< 3, 4, 6 > . \end{aligned} \tag{2.5}$$

Množici povezav za oba modula sta

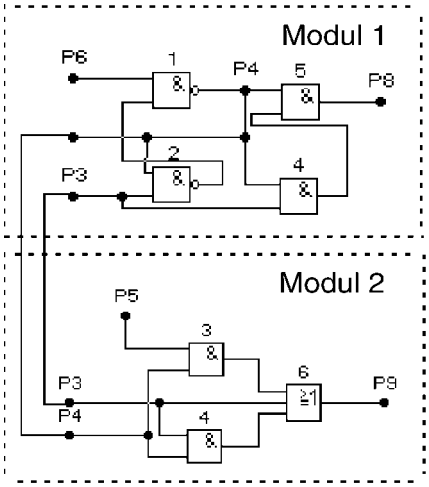
$$\begin{aligned} (M1)^* &= (1 \cup 2 \cup 4 \cup 5) =< P2, P3, P4, P6, P8 > \\ (M2)^* &= (3 \cup 4 \cup 6) =< P2, P3, P4, P5, P9 > . \end{aligned} \tag{2.6}$$

V množici povezav $(M1)^*$manjka notranja povezava $P1$ in v množici $(M2)^*$ notranja povezava $P7$. Na sliki 2.13 je razvidno, da je možno iz povezav $P3$ in $P4$ priti do povezave $P2$ prek operatorja 4, $P2 = f(P3, P4)$. To se lahko zgodi v obeh modulih, saj se modul 4 nahaja v obeh modulih. Na ta način ima povezava $P2$ povsem notranji pomen in jo med moduloma ni potrebno vpeljati. Iz izrazov tako dobimo množici povezav

$$\begin{aligned} (M1) &=< P3, P4, P6, P8 > \\ (M2) &=< P3, P4, P5, P9 > \end{aligned} \tag{2.7}$$

ki ju omogoča replikacija operatorja 4. Nemodularna preklopna struktura na sliki 2.13 je na osnovi izrazov (2.7) prevedena na modularno strukturo na sliki 2.14. Izpolnjen je pogoj omejitev, ki zahtevajo največ 4 operatorje na en modul in največ 4 medmodulske povezave na en modul.

Pomembno je, da je v IC-tehnologiji največkrat priključek na čipu pomembnejši (dražji) od logičnega operatorja. Prav zato ima replikacijska metoda svoj smisel in veljavo.

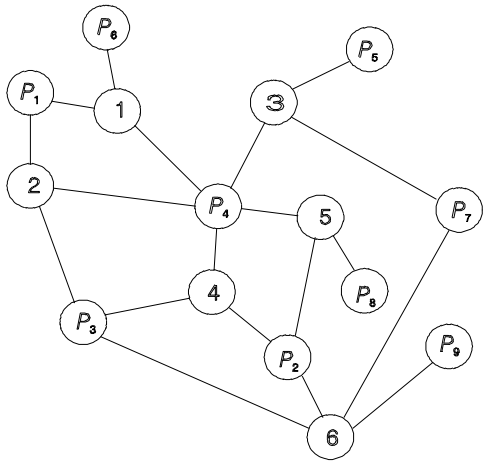


Slika 2.14: Modularna zgradba logične sheme na sliki 2.13.

2.4.3 Metoda pokritja

Preklopno vezje na sliki 2.13 si narišimo v obliki bigrafa, kot to prikazuje slika 2.15. Množice povezav posameznih logičnih operatorjev smo si že izpisali z izrazi (2.3). Če imamo omejitve 4 operatorje/modul in 4 med-modulske povezave/modul, ne moremo postaviti več kot dva operatorja. Zadnje izhaja iz danega bigrafa in ustreznega spiska vseh možnih združevanj logičnih operatorjev, kot sledi:

$(1) =< P1, P4, P6 >$
$(2) =< P1, P3, P4 >$
$(3) =< P4, P5, P7 >$
$(4) =< P2, P3, P4 >$
$(5) =< P2, P4, P8 >$
$(6) =< P2, P3, P7, P9 >$
$(1) \cup (2) =< P1, P3, P4, P6 >$
$(1) \cup (3) =< P1, P4, P5, P6 >$
$(1) \cup (4) =< P1, P2, P3, P4, P6 >$
$(1) \cup (5) =< P1, P2, P4, P6, P8 >$
$(1) \cup (6) =< P1, P2, P3, P4, P6, P7, P9 >$
$(2) \cup (3) =< P1, P3, P4, P5, P7 >$
$(2) \cup (4) =< P1, P2, P3, P4 >$



Slika 2.15: Bigraf za logično funkcijo na sliki 2.13.

$(2) \cup (5) =< P1, P2, P3, P4, P8 >$
$(2) \cup (6) =< P1, P2, P3, P4, P7, P9 >$
$(3) \cup (4) =< P2, P3, P4, P5, P7 >$
$(3) \cup (5) =< P2, P4, P5, P7, P8 >$
$(3) \cup (6) =< P2, P3, P4, P5, P7, P9 >$
$(4) \cup (5) =< P2, P3, P4, P8 >$
$(4) \cup (6) =< P2, P3, P4, P7, P9 >$
$(5) \cup (6) =< P2, P3, P4, P7, P8, P9 >$
$(1) \cup (2) \cup (3) =< P1, P3, P4, P5, P6, P7 >$
$(1) \cup (2) \cup (4) =< P1, P2, P3, P4, P6 >$
$(1) \cup (2) \cup (5) =< P1, P2, P3, P4, P6, P8 >$
$(1) \cup (2) \cup (6) =< P1, P2, P3, P4, P6, P7, P9 >$
$(1) \cup (3) \cup (4) =< P1, P2, P3, P4, P5, P6, P7 >$
$(1) \cup (3) \cup (5) =< P1, P2, P4, P5, P6, P7, P8 >$
$(1) \cup (3) \cup (6) =< P1, P2, P3, P5, P6, P7, P9 >$
$(1) \cup (4) \cup (5) =< P1, P2, P3, P4, P6, P8 >$
$(1) \cup (4) \cup (6) =< P1, P2, P3, P4, P6, P7, P9 >$
$(1) \cup (5) \cup (6) =< P1, P2, P3, P4, P6, P7, P8, P9 >$
$(2) \cup (3) \cup (4) =< P1, P2, P3, P4, P5, P7 >$
$(2) \cup (3) \cup (5) =< P1, P2, P3, P4, P5, P7, P8 >$
$(2) \cup (3) \cup (6) =< P1, P2, P3, P4, P5, P7, P9 >$

$$(2) \cup (4) \cup (5) =< P1, P2, P3, P4, P8 >$$
$$(2) \cup (4) \cup (6) =< P1, P2, P3, P4, P7, P9 >$$
$$(2) \cup (5) \cup (6) =< P1, P2, P3, P4, P7, P8, P9 >$$
$$(3) \cup (4) \cup (5) =< P2, P3, P4, P5, P7, P8 >$$
$$(3) \cup (4) \cup (6) =< P2, P3, P4, P5, P7, P9 >$$
$$(3) \cup (5) \cup (6) =< P2, P3, P4, P5, P7, P8, P9 >$$
$$(4) \cup (5) \cup (6) =< P2, P3, P4, P7, P8, P9 >$$

$$(1) \cup (2) \cup (3) \cup (4) =< P1, P2, P3, P4, P5, P6, P7 >$$
$$(1) \cup (2) \cup (3) \cup (5) =< P1, P2, P3, P4, P5, P6, P7, P8 >$$
$$(1) \cup (2) \cup (3) \cup (6) =< P1, P2, P3, P4, P5, P6, P7, P9 >$$
$$(1) \cup (2) \cup (4) \cup (5) =< P1, P2, P3, P4, P6, P8 >$$
$$(1) \cup (2) \cup (4) \cup (6) =< P1, P2, P3, P4, P6, P7, P9 >$$
$$(1) \cup (2) \cup (5) \cup (6) =< P1, P2, P3, P4, P6, P7, P8, P9 >$$
$$(1) \cup (3) \cup (4) \cup (5) =< P1, P2, P3, P4, P5, P6, P7, P8 >$$
$$(1) \cup (3) \cup (4) \cup (6) =< P1, P2, P3, P4, P5, P6, P7, P9 >$$
$$(1) \cup (3) \cup (5) \cup (6) =< P1, P2, P3, P4, P5, P6, P7, P8, P9 >$$
$$(1) \cup (4) \cup (5) \cup (6) =< P1, P2, P3, P4, P6, P7, P8, P9 >$$
$$(2) \cup (3) \cup (4) \cup (5) =< P1, P2, P3, P4, P5, P7, P8, P9 >$$
$$(2) \cup (3) \cup (4) \cup (6) =< P1, P2, P3, P4, P5, P7, P9 >$$
$$(2) \cup (4) \cup (5) \cup (6) =< P1, P2, P3, P4, P7, P8, P9 >$$
$$(3) \cup (4) \cup (5) \cup (6) =< P2, P3, P4, P5, P7, P8, P9 >$$

$$(\textit{konec porojevanja unij}) \tag{2.8}$$

Porojevanje unij pred dolžino 5 prekinemo, ker obstaja dana omejitev: največ 4 operatorje/modul. Nobena od zgoraj navedenih kombinacij operatorjev, ki ima unijo dolžine 3 ali 4 nam ne ustreza, ker je povsod že preseženo število medmodulskih povezav 4 (desna stran izrazov). Rešitev lahko iščemo le v okviru tistih izrazov (2.8), ki imajo na levi strani unijo dolžine 1 ali 2, ker s tem na desni strani število elementov v množici še ne presega števila 4. Za nadaljnji postopek pridejo torej v poštev le izrazi:

	Minterm					
Glavni vsebovalnik	1	2	3	4	5	6
(1)	$\vee$					
(2)		$\vee$				
(3)			$\surd$			
(4)				$\vee$		
(5)					$\vee$	
(6)						$\surd$
$(1) \cup (2)$	$\surd$	$\surd$				
$(2) \cup (4)$		$\vee$		$\vee$		
$(4) \cup (5)$				$\surd$	$\surd$	

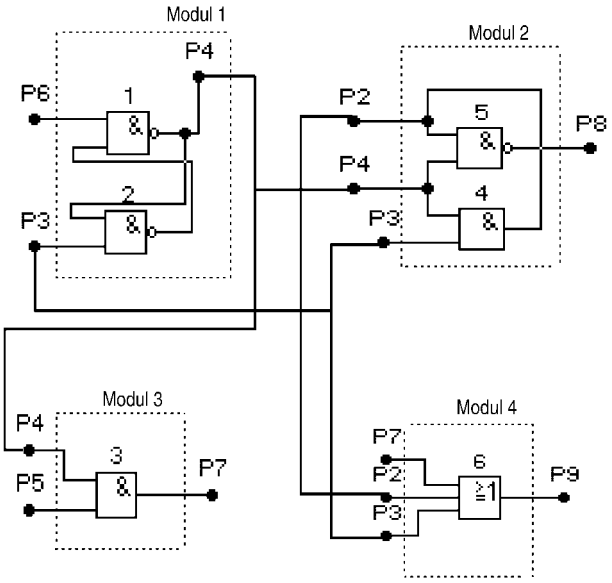
Tabela 2.1: Določanje potrebnih glavnih vsebovalnikov.

$$\begin{aligned}
&(1) =< P1, P4, P6 > \\
&(2) =< P1, P3, P4 > \\
&(3) =< P4, P5, P7 > \\
&(4) =< P2, P3, P4 > \\
&(5) =< P2, P4, P8 > \\
&(6) =< P2, P3, P7, P9 > \\
&(1) \cup (2) =< P1, P3, P4, P6 > \\
&(2) \cup (4) =< P1, P2, P3, P4 > \\
&(4) \cup (5) =< P2, P3, P4, P8 > .
\end{aligned} \tag{2.9}$$

Optimalna rešitev izhaja iz Quineove metode poenostavljanja preklopnih funkcij [1], če le privzamemo, da so izrazi (2.9) glavni vsebovalniki oziroma izolirane konjunkcije in mintermi operatorji $i = 1, 2, ..., 6$. Iz tabele 2.1 izhaja, da je možno minimalno pokritje

$$(1) \cup (2), (3), (4) \cup (5), (6), \tag{2.10}$$

v katerem je vsebovalnik $(2) \cup (4)$, ki je nepotreben v minimalni disjunktivni normalni obliki, tudi nepotreben v optimalni modularni shemi. Veliki znak vsebovanja $\surd$ pokriva vse stolpce (vse operatorje dane logične sheme), kar pomeni, da je predlagani izbor (2.10) rešitev problema. Ustrezno modularno shemo vidimo na sliki 2.16. V njej so 4 moduli, v vsakem pa sta



Slika 2.16: Modularna zgradba logične sheme na sliki 2.13.

največ 2 operatorja. Medmodulski povezavi sta $P2$, in $P4$, medtem ko je povezava $P3$ tako ali tako vhodna.

Primerjava shem na slikah 2.13 in 2.16 kaže, da je replikacijska metoda uspešnejša od metode pokritja, saj imamo v primeru prve 2 modula in v primeru druge kar 4 module. Vendar gremo lahko tudi v primeru izrazov (2.8) v replikacijo, kar vodi v kombinirano metodo. Za razlago tega pristopa izločimo iz (2.8) naslednja ugodna izraza

$$\begin{array}{l}(1) \cup (2) \cup (4) \cup (5) =< P1, P2, P3, P4, P6, P8 > \\ (3) \cup (4) \cup (6) =< P2, P3, P4, P5, P7, P9 > .\end{array} \tag{2.11}$$

Na osnovi replikacije lahko napravimo naslednje funkcije

$$\begin{array}{l}P1 = f1(P3, P4) \\ P2 = f2(P3, P4) \\ P7 = f3(P4, P5).\end{array} \tag{2.12}$$

kar pomeni, da v (2.11) odpadejo notranje povezave $P1$, $P2$ in $P7$. Tako je logična shema lahko postavljena z dvema moduloma

$$\begin{array}{l}(1) \cup (2) \cup (4) \cup (5) =< P3, P4, P6, P8 > \\ (3) \cup (4) \cup (6) =< P3, P4, P5, P9 > .\end{array} \tag{2.13}$$

Oba zadnja dva izraza sta glede omejitev dovoljena. Prišli smo tako do iste učinkovitosti, kot jo imamo pri replikacijski metodi. Izraza (2.13) kažeta na to, da sta množici $(1) \cup (2)$ in $(4) \cup (5)$ vendarle združljivi, združljivi pa sta tudi (3) in (4) - vendar pa ne brez funkcij (2.12).

2.4.4 Metoda matematičnega programiranja

Vzemimo, da ima velika logična struktura n logičnih operatorjev (lahko tudi že blokov takšnih operatorjev). Če želimo te razmestiti na k podmnožic (modulov) tako, da je v vsaki podmnožici m operatorjev, imamo vseh možnosti za nameščanje

$$Y = \frac{n!}{k!\,(m!)^k}. \tag{2.14}$$

Zadnje predstavlja tako veliko število, da je potrebno posegati po hevrističnih in izkustvenih postopkih načrtovanja.

Zgled 2 *:*

Ocenimo število možnih namestitev, če je $n = 1000$, $k = 20$ *in* $m = 50$.

$$Y = \frac{1000!}{20!\,(50!)^{20}} > 10^{1260}.$$

Pri tako velikih številih Y *strokovnjak izgubi možnost in smisel za kakršnokoli načrtovanje, čeprav je opazovani velikostni red za* n, k *in* m *nekaj možnega v računalniških sistemih.*

Da vidimo, kakšna je realna možnost formalnega, to je matematičnega reševanja problemov pri razmeščanju komponent RSS po modulih, si poglejmo algoritem ki sloni na linearnem programiranju.

Pri razmeščanju lahko dva operatorja i in j damo ali v isti modul M ali pa ju damo v dva različna modula M_a in M_b. Problem takšnega razmeščanja obvladamo na osnovi naslednjih spremenljivk:

- x_{ij}, za določanje prisotnosti/odsotnosti operatorjev v opazovanem modulu,
- c_{ij}, število povezav med operatorjem i in operatorjem j,
- w_{ij}, utež, da operator i spada v isti modul kot operator j,
- m, število operatorjev na modul (jemljemo kot omejitev) in
- p, število priključkov na modul (jemljemo kot omejitev).

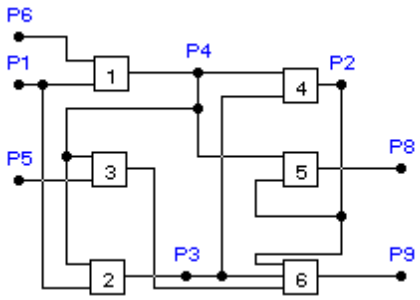
Možne vrednosti za podane spremenljivke operatorjev i in j so:

$$\begin{array}{l} x_{ij} = 1, \text{če je } i \text{ v drugem modulu kot } j \\ x_{ij} = 0, \text{če je } i \text{ v istem modulu kot } j \\ \\ c_{ij} = c_{ji} = 0, \text{ ni povezave med } i \text{ in } j \\ c_{ij} = c_{ji} = k, \text{število povezav med med } i \text{ in } j \\ k \text{ je celo pozitivno število} \\ \\ w_{ij} = w_{ji} = 1, \text{ ni posebne afinitete med } i \text{ in } j \\ w_{ij} = w_{ji} = h, \text{ stopnja afinitete med } i \text{ in } j \\ h \text{ je celo pozitivno število.} \end{array} \tag{2.15}$$

Preklopna spremenljivka x_{ij} pove s svojo vrednostjo ali gre za medmodulsko ali modulsko sodelovanje operatorjev i in j. Spremenljivka c_{ij} pove, koliko povezav veže dva operatorja, zaenkrat še ne glede na to ali so povezave interne, modulske, ali medmodulske. Kot vidimo, imamo poleg spremenljivk x in c še utež w_{ij}, ki pove, ali ima operator i kakšno posebno afiniteto do tega, da leži z j v istem modulu (na primer pomembna je izredno kratka dolžina povezav med i in j).

Pri danih spremenljivkah je možno postaviti naslednji sistem neenačb:

$$\begin{array}{l} \sum_{j=1}^{n} w_{ij}(1 - x_{ij}) \leq m \quad \text{za vsak } i = 1, 2, ..., n, \ j \neq i \\ \\ \sum_{j=1}^{n} c_{ij} x_{ij} \leq p \quad \text{za vsak } i = 1, 2, ..., n, \ j \neq i \\ \\ x_{ij} + x_{jk} + x_{ik} \neq 1 \, . \end{array} \tag{2.16}$$



Slika 2.17: Primer velike logične strukture (šolski primer).

Prva vsota predpostavlja, da je v modulu omejitev na število operatorjev m, druga pa, da je največ medmodulskih povezav na modul toliko kot priključkov p. Če bi imeli v tretjem gornjem izrazu enakost, bi morala biti dva operatorja hkrati v istem modulu in izven njega, kar seveda ni mogoče (pravilo tranzitivnosti).

Sedaj, ko imamo postopek, si poglejmo razmeščanje komponent logične sheme na sliki 2.17 pri podatkih $m = 4$, $p = 5$, $w_i = 1$, $i = 1, 2, ..., 6$. Glede na (2.16) imamo opraviti s spremenljivkami $x_{ij} = x_{ji}$ pri $i, j = 1, 2, ..., 6$.

Da ne bomo ročno optimizirali razdeljevanje operatorjev na module, si na osnovi (2.16) najprej napravimo ustrezni program.

V tehniki linearnega programiranja obstaja pojem objektivne funkcije. Na to funkcijo navežimo neenačbe (2.16) tako, da nam optimizacija te funkcije da vrednosti vseh x_{ij}, ki predstavljajo optimalno namestitev logičnih operatorjev v module opazovane logične sheme. Izberimo orodje MATLAB V5, v okviru katerega obstaja optimizacijski paket Optimization Toolbox [3]. V začetku Dodatka v tem učbeniku je podan program LPR.M, ki smo ga napisali z namenom, da prikažemo aplikacijo metode matematičnega programiranja v primeru logične sheme na sliki 2.17. Iz programa izhaja, da ta potrebuje funkcijo LINPRO.M, ki je podana na koncu omenjenega programa.

V programu je uporabljena rutina `constr( )`, ki da optimalno rešitev. Da pridemo do te rešitve, moramo procesirati funkcijo LINPRO(**x**). Objektivna funkcija je

$$f(\mathbf{x}) = \sum_{i=1}^{15} x(i)\,.$$

Pri tem je i (v programu) tekoči indeks spremenljivk x_{12} do x_{56}, ki te spremenljivke razvrsti v vektor $\mathbf{x}$ (glej program LPR.M). Da dobimo čim več logičnih operatorjev na en modul, mora biti čim več spremenljivk $x_{ij} = 0$. Prav zato je za nas smiselna takšna optimizacija, pri kateri je funkcija f minimizirana. Matrika $\mathtt{g(r,1)}$ je matrika omejitev (2.16). Neenačbe so zapisane tako, da so $\mathtt{g(r,1)} \leq 0$. Začetni vektor je $\mathbf{x}_0 = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)$. Zadnje predvideva, da so vsi logični elementi v začetku v enem modulu. Seveda to z realnega stališča gledano ni možno, saj imamo opraviti z omejitvami m in p. Funkcijska vrednost je pri začetnem vektorju $f(\mathbf{x}_0) = 0$, vendar, če naj veljajo omejitve m in p, mora biti ta vrednost večja od 0 sicer pa minimalna, da pridemo do čim manj modulov in čim manj medmodulskih povezav. Razsežnost omejitev je v LINPRO($\mathbf{x}$) postavljena v okvir spremenljivke `g(1:30,1)`. Vsaka omejitev je podana v dveh možnih oblikah, izbrati pa moramo samo eno od obeh. Zakaj je temu tako? Tretji izraz v (2.16) zahteva različnost $\neq$, kar pa definicija neenačbe, na kateri sloni linearno programiranje, ne dopušča. Možno je imeti $>$, $<$, $=$, $\leq$, ali $\geq$, ni pa možno $<>$, $><$, ali $\neq$. Nastali problem rešimo tako, da nedopustno ralacijo razdrobimo v dve dopustne, med njima pa uberemo dvojiško izbiranje. Možnih poskusov minimiziranja je tako 230. Zadnje je zelo veliko primerkov, da bi med njimi iskali povsem dokončno, to je formalno rešitev. Lahko napravimo manj kot 230 poskusov, če nam to da, strokovno gledano, zadovoljiv rezultat.

Naslednji problem je v tem, ker postopki linearnega programiranja v računalniških orodjih (na primer MATLAB, Mathematica) zahtevajo, da so spremenljivke x_{ij} normalne spremenljivke (predvidene so ali cele ali realne vrednosti spremenljivk), ne pa logične (preklopne), ki so uporabljene v naši metodi. Odločimo se za postopek optimizacije pri realnih vrednostih omenjenih spremenljivk. Da na nek način vendarle po določitvi takšnega optimuma pridemo do napotkov za razdelitev logičnih operatorjev po algoritmu (2.16), napravimo v programu LPR.M naslednjo aproksimacijo: vse vsednosti, ki so manjše od 0.5 nadomestimo z 0 in vse vrednosti, ki so večje od 0.5 pa z 1. Seveda moramo pri tem kontrolirati, ali se nam značilnost minimuma objektivne funkcije vsaj do določene mere ohranja.

Rezultati opisanega programa so naslednji:

```
x = [1.3899 0.1101 0.4519 1.7316 -0.8418 0.4519 0.1101
    -0.5140 -0.5000 -0.5140 0.1101 2.6836 0.4519
     0.4379 -0.8418]
x1 = [1 0 0 1 0 0 0 0 0 0 0 1 0 0 0]
f = 4.7177
f1 = 3
g = [-1.8418 0.0621 -1.8418 0.0621 0.0621 0.0621
      0.0734 -2.0481 -2.1582 -3.0621 -3.6101
     -2.7823 0.0481 0.0481 -0.6075 0.0481 0.0481
      0.0481 0.0481 -0.6355 0.0481 0.0481 0.0481
     -0.6355 0.0481 0.0481 -1.8558 0.0481 -0.6075
      0.0621]
```

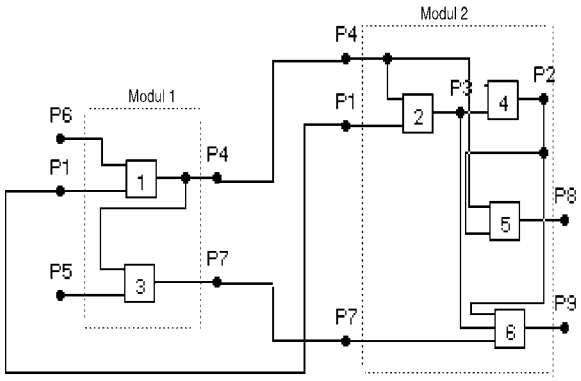
Minimum objektivne funkcije, ki ga dobimo pri vektorju $\mathbf{x}$, je $f(\mathbf{x}) = 4.7177$. Iz $\mathbf{x}$ dobimo dvojiški vektor $\mathbf{x}_1$, ki nam da vrednost objektivne funkcije $f(\mathbf{x}_1) = 3$, torej manj kot pri realnem optimumu. Zadnje pomeni, da izbrana dvojiška koda za omejitve `g` še ni bila optimalno izbrana. Če bi izbirali nadaljnje kode, bi prišli do optimuma, ki bi bil nižji od 3 ali vsaj enak 3. Pri opravljenem iskanju primerne dvojiške kode za `g` smo zmanjšali vrednost za `f` iz 13 na 4.7177. Ne da bi to vrednost še zmanjševali, se zadovoljimo z rezultatom. Kaj smo dobili? Z indeksnim prehodom nazaj na spremenljivke x_{ij} imamo pri vektorju $\mathbf{x}_1$

$$x_{13} = x_{14} = x_{16} = x_{23} = x_{24} = x_{25} = 0$$
$$x_{26} = x_{34} = x_{35} = x_{45} = x_{46} = x_{56} = 0$$
$$x_{12} = x_{15} = x_{36} = 1\,.$$

Z uporabo zakona tranzitivnosti nam te vrednosti dajo dva modula z logičnimi operatorji: $M_a = \{2, 4, 5, 6\}$ in $M_b = \{1, 3\}$. Modul M_a lahko izberemo, ker velja pri vseh njegovih parih $(2, 4), (2, 5), (2, 6), (4, 5), (4, 6)$, (5,6) vrednost $x_{ij} = 0$. Na sliki 2.18 imamo podana oba modula, ki skupaj ustrezata dani shemi na sliki 2.17.

2.4.5 Metoda minimalne zakasnitve v RSS

Do sedaj obravnavane metode razmeščanja operatorjev na module predpostavljajo, da želimo priti do RSS, ki ima čim manjše število modulov (veliko vsebino na modul) in čim manjše število povezav med njimi. Povsem drug pristop pa imamo pri razmeščanju, če so za nas pomembne zakasnitve signala, ki potuje skozi RSS. V tem primeru želimo minimizirati časovne zakasnitve, ki obstajajo med vhodnimi in izhodnimi sponkami RSS.



Slika 2.18: Modularna shema za logično shemo na sliki 2.17.

Medmodulska povezava je dolga, in je zato v nsek-področju že lahko kritična. Nasprotno pa je notranja povezava (v modulu) kratka, kar pomeni, da je za časovne zakasnitve z vidika zunanjega opazovalca, nekritična.

Pri preučevanju zakasnitev RSS je zelo pomembno, kakšen je matematični graf RSS. Svoj smisel v grafih dobi smer signalov, izvornost in ponornost signalov, skratka, grafi so usmerjeni. Predvsem je pomebna razlika med cikličnim ali necikličnim grafom. Pri cikličnem grafu imamo povratne povezave, v necikličnem primeru pa takšnih povezav nimamo. Postopki za minimizacijo zakasnitev v necikličnem primeru so enostavnejši in krajši.

Ponorno drevo je graf, ki je predstavljen z več vhodnimi, vmesnimi in enim izhodnim vrhom. Z vidika signalov je ta graf usmerjen, saj potekajo signali preko logičnih operatorjev sheme od vhodnih vrhov k izhodnemu vrhu sicer pa ta usmerjenost grafa za postavljanje modulov ne igra posebne vloge. Izhodni vrh drevesa ustreza vedno izhodu zadnjega logičnega operatorja v shemi, vhodni vrhovi pa vhodom prvih logičnih operatorjev v shemi. Vrh i v grafu ima svojo utež w_i in svoje predhodnike, ki so nanizani v smeri vhodov. V isti modul gredo lahko vsi predhodniki, če je teh manj od števila logičnih elementov/modul m sicer pa jih nekaj ostane zunaj in gredo zato v naslednji modul. Elementi, ki so v istem modulu, imajo enako zakasnitev sicer pa so zakasnitve različne. Ime $i(k)$ pomeni, da vrh, ki ustreza operatorju i v opazovani logični shemi, sodi v modul z zakasnilnim nivojem k. Na vhodu grafa je nezakasnitev (logične sheme), njen nivo zakasnitve je torej (0). Pri tem je $i(0)$ ime vrha, ki je na vhodu grafa, kjer

še ni nobene zakasnitve. V grafu je lahko več modulov istega zakasnilnega nivoja (k), zato ustrezno temu damo modulom različna imena.

Postopek, ki da pri ponornem drevesu najmanjšo zakasnitev, je naslednji:

1. Vse primarne vhodne vrhove označimo z imenom 0. V prve module damo toliko teh vrhov, da je izpolnjena omejitev m.

2. Vrhu i, ki ima predhodnike $0, 1, 2, ..., i-1$ in je pri njem izpolnjen pogoj

$$w_i + \sum_{j=0,\,1,\,...,i-1} w_j \leq m \tag{2.17}$$

   damo v modul k, če so le predhodniki v k modulu. V nasprotnem primeru damo vrh i v modul $k+1$.

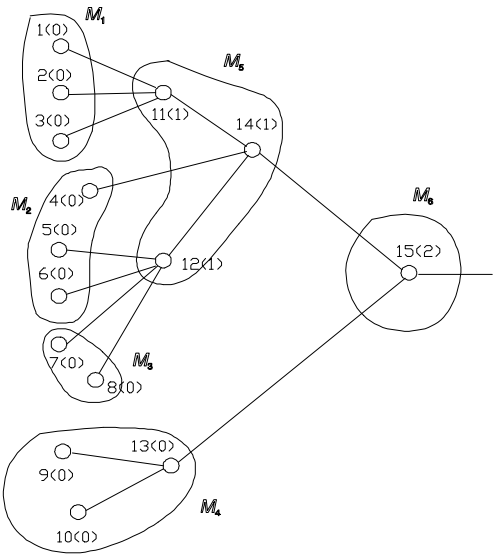
3. V modulu k so vrhovi povezani v nizu ali pa tudi ne. Lahko so celo nepovezani (vhodni moduli). Važno je, da je $\sum w_i \leq m$.

4. Zakasnitev RSS je določena z imenom največjega nivoja zakasnitve, ki je na izhodni strani grafa. Ta zakasnitev je najmanjša glede na pogoj m.

Ta postopek si oglejmo v okviru grafa na sliki 2.19. Vsak vrh je označen s svojo številko in nivojem zakasnitve, v katerega spada. Postopek ugotavlja najmanjšo zakasnitev pri pogoju $m = 3$ in $w_i = 1$, $i = 1, 2, ..., 15$. V principu ob upoštevanju omejitve m "oblačimo" vrhove v grafu tako, da začnemo na skrajni levi in pri tem ne izpustimo nobenega vrha. Iz rezultata izhaja, da je v modularni shemi zakasnitev 2, medtem ko je v nemodularni (dani) shemi zakasnitev 3. V prvem primeru imamo od izvora do ponora največ 2 medmodulski povezavi, medtem ko imamo v drugem primeru največ 3 medmodulske povezave.

Izvorno drevo je graf, v katerem se vrhovi razcepljajo iz enega vhodnega vrha. Postopek za postavitev modulov v takšnem grafu je enak kot pri ponornem drevesu, le da z označevanjem nivojev zakasnitve in modulov pričnemo na nasprotni strani, v našem primeru na desni. Če smo pri ponornem drevesu z (0) označevali nivo zakasnitve vhodov, pri izvornem drevesu označujemo z (0) izhodni nivo zakasnitve.

Splošen neciklični graf se hkrati nanaša na izvornost in ponornost grafa. Ker smo obdelali vsak primer zase, tudi s splošnim grafom ne bi smeli

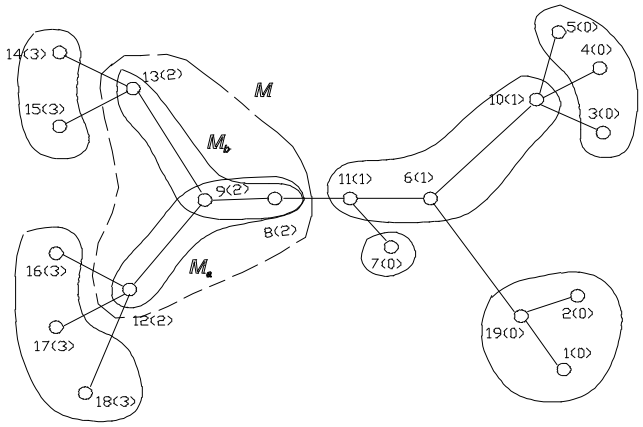


Slika 2.19: Postavljanje modulov glede na minimalno zakasnitev.

imeti težav. Temu ni tako le na mestu, kjer ponornost prehaja v izvornost in narobe, izvornost v ponornost. Rešitev problema omogoča replikacija vrhov, ki jo uvedemo na omenjenih dvojnih mestih.

Oglejmo si določitev minimalne zakasnitve "z desne strani" za graf na sliki 2.20. Vrhova 8 in 9 v postopku enkrat upoštevamo proti vrhu 13 in enkrat proti vrhu 12. V prvem primeru so v modulu operatorji $\{8, 9, 13\}$, v drugem pa $\{12, 8, 9\}$. Upoštevanje z obeh strani nam da modul z operatorji $\{8, 9, 12\} \cup \{12, 8, 9\} = \{8, 9, 12, 13\}$. Ta modul s štirimi operatorji ni dopusten, ker imamo omejitev $m = 3$. Nastalo težavo na prehodu iz porojevanja v ponornost in narobe odstranimo z replikacijo vrhov s številkama 8 in 9. Na sliki 2.20 pridemo tako iz enega nepravilno postavljenega modula M na dva, M_a in M_b, ki sta pravilno postavljena, vsebujeta pa vsak zase tudi operatorja 8 in 9.

Do sedaj smo pri zakasnitvi skozi RSS opazovali le omejitev m, to je število operatorjev na modul. Če k tej omejitvi dodamo tudi p kot omejitev v številu priključkov (medmodulskih povezav) na modul, se nam postopek nekoliko podaljša oziroma zamota. Poglejmo si še to posplošitev! Naj je V v splošnem množica povezav modula. Množica $V(i, k)$ naj združuje vse



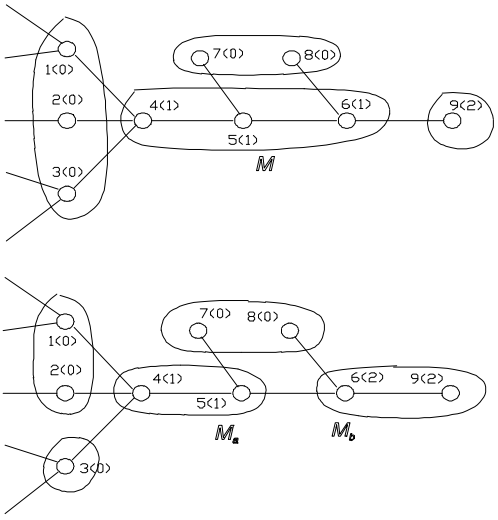
Slika 2.20: Izvorno-ponorni graf označen z $m = 3$ in $w_i = 1$.

vrhove grafa (logične operatorje), ki so predhodniki vrhu i in imajo nivo zakasnitve k. Število povezav ali priključkov, ki jih ima modul glede na druge module, naj je $p(V)$. Prejšnji algoritem za razmeščanje po omejitvi m (glej izraz (2.17)) se spremeni takole

$$\begin{aligned} w_i + \sum_{j \in V(i,k)} w_j &\leqslant m \\ p(V(i,k) \cup \{i\}) &\leqslant p \end{aligned} \tag{2.18}$$

Spremenila se je torej spodnja meja za indeks j, drugi izraz pa je dodan, saj priključkov p v prejšnejm postopku nismo obravnavali. $\{i\}$ je množica povezav, ki jih ima zadnji opazovani vrh i nasproti ostalim modulom sheme. Če sta izpolnjena oba gornja pogoja, damo vrhu s številko i nivo zakasnitve k, v nasprotnem primeru pa $k + 1$.

Za dodatno razlago si poglejmo sliko 2.21 pri omejitvah $m = 3$ in $p = 5$. V gornjem delu omenjene slike imamo razporeditev modulov, ki zadošča samo omejitvam m. Modul M z zakasnilnim nivojem (1) ustreza omejitvi m, ne ustreza pa omejitvi p, saj ima 6 povezav z drugimi moduli. Da bi prišli do manjšega števila priključkov na modul M, napravimo test po zgoraj omenjenem postopku. Za vrh $i = 6$, katerega opazujemo ali sodi v M z (1) ali v naslednji modul, imamo mnnožico $V(i,k) = V(6,1) = \{4,5\}$.



Slika 2.21: Razdeljevanje operatorjev na module ob upoštevanju zakasnitev in omejitev m ter p.

Velja $p(V(6,1)) = 4$ in $\{i\} = \{6\}$. Ker je $p(\{6\}) = 2$, imamo opraviti z relacijo za priključke

$$p(V(6,1) \cup \{6\}) = 4 + 2 = 6 > p.$$

Ker je meja $p = 5$ presežena, je novi nivo zakasnitve $(k') = (k + 1) = 2$. Ustrezno spremembo v razporejanju vrhov v module vidimo na sliki 2.21 spodaj[1].

[1] Žargon v tem delu nas včasih nekoliko zanese v nenatačno izražanje. Na primer, vrhov grafa ne razporejamo v module, tako smo ravnokar rekli - v module razporejamo logične operatorje, to je elemente iz logične sheme. Z vidika matematike bi bilo boljše, da bi rekli: vrhove razporejamo v skupke vrhov (clasters). Vendar pa, ker je med skupkom vrhov in ustreznim modulom logičnih elementov transparentna preslikava, lahko pojma v hitrem načrtovalskem žargonu zamenjujemo med seboj. Da nam ne bi bilo potrebno vnašati še dodatne definicije, nas omenjeni pristop ne sme preveč razočarati.

2.5 NAMESTITEV MODULOV NA KARTICO

2.5.1 Uvod

V prejšnjih podpoglavjih smo obravnavali postopke, ki poskušajo veliko RSS razdeliti na module tako, da imamo na koncu opraviti z modularno shemo, ki naj bi bila po predpostavljenih kriterijih optimalna. Spoznali smo, da moramo nameščene module medsebojno povezovati z medmodulskimi povezavami, če hočemo priti do realizacije opazovane RSS. Za vsak modul imamo podano število priključkov in število ter vrsto operatorjev, ki sodijo v modul. Za realizacijo RSS pa to ni dovolj, saj nimamo še nobenega podatka o tem, kako so moduli fizično nameščeni na kartico. Pri tipiziranih gradnjah imamo N možnih pozicij, medtem ko ima opazovana RSS M modulov. V splošnem je možno le $M \leq N$. Katerikoli modul lahko damo na katerokoli pozicijo. Vsaka takšna namestitev pa ima lahko dobre in tudi slabe značilnosti za delovanje RSS.

V splošnem bi morali pri namestitvi modulov na kartico (ali na drug podoben medij) paziti na vrsto parametrov oziroma omejitev, kot so: topološke omejitve, razdalje med karticami, električne omejitve, hitrost delovanja kartice, lega vodil, vzemljitve itd. Zahtevam v veliki meri zadostimo, če optimiziramo totalno dolžino povezav med vsemi moduli, ki jo prestavlja seštevek vseh posameznih medmodulskih povezav. Pri najmanjši totalni dolžini povezav so tudi kratke posamezne povezave, majhni vplivi zakasnitev, refleksov, presluha itd. Najkraša povezava največkrat nosi s seboj tudi minimalne stroške (minimal costs). Prav zato so postopki za optimalno nameščanje modulov na kartice največkrat in najbolj vezani na totalno dolžino vseh medmodulskih povezav RSS na kartici. Tudi mi bomo totalni dolžini povezav posvetili kar nekaj prostora.

2.5.2 Spodnje meje totalne dolžine povezav na kartici

Število vseh permutacij v primeru M modulov pri N pozicijah na kartici podaja izraz

$$P = \left(\begin{array}{c} N \\ M \end{array} \right) M! \tag{2.19}$$

Pri tem lahko pride vsak modul na vsako pozicijo. Če ima modul M_a q povezav do modula M_b, ima tudi modul M_b q povezav do modula M_a. Zadnje pomeni, da smer informacije v povezavi pri fizični postavitvi modulov

na kartico ne igra nobene vloge. Ker imamo omenjeno simetričnost, pade število P na polovico, saj velja enakovrednost postavitev $(1, 2, ..., N)$ in $(N, N-1, ..., 1)$. Do enakovrednih postavitev pridemo tudi v primeru, ko pri koordinatnem nameščanju (koordinatna mreža) zamenjamo navpičnost z vodoravnostjo in narobe.

Če je N veliko število, je potrebno posegati po raznih približnih ocenah, med katere sodi tudi splošna spodnja meja totalne dolžine povezav (SMTDP). Ko poznamo to oceno dolžine, še ne pomeni, da ima resnično postavljena permutacija (namestitev) dolžino enako SMTDP. Vemo le to, kakšna je najmanjša možna spodnja vrednost, ki jo lahko zavzame totalna dolžina povezav (TDP) ne glede na permutacijo. Možno pa je seveda določiti tudi SMTDP, ki pripada točno določeni permutaciji. Tej dolžini bomo rekli SMTDP na permutacijo (SMTDPP).

Vzemimo, da imamo na kartici modula i in j tako nameščena, da obstaja med njima razdalja d_{ij}. Ne glede na to kje sta nameščena na kartici, obstaja med njima c_{ij} medsebojnih povezav. Na osnovi teh postavk lahko nastavimo matriki

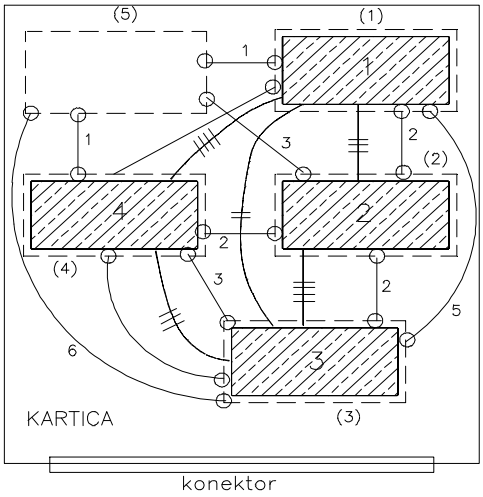
$$C = [c_{ij}] \quad D = [dij]\,. \tag{2.20}$$

C je matrika povezav in D matrika razdalj med pozicijami. Da pridemo do najkrajše totalne dolžine povezav, moramo minimizirati (stroškovno, objektivno) funkcijo

$$f(c, d) = \sum c_{ij} d_{ij}\,. \tag{2.21}$$

V primeru $M < N$ ima matrika povezav C manjši red od matrike razdalj D. Obe matriki izravnamo po redu matrike D s tem, da stolpce in vrstice v matriki C ustrezno dopolnimo z ničlami. S tem pridemo do možnosti, da nad matrikama lahko izvajamo operacije.

Za razlago določanja SMTDP si oglejmo kartico na sliki 2.22. Na njej imamo $N = 5$ pozicij in $M = 4$ module, kar pomeni, da gre za slučaj $M < N$. Prazna je pozicija (5), saj na njej ni postavljen noben modul. Na ostalih pozicijah (i) imamo postavljene module i. Število povezav med dvema modulama je podano s pravokotnimi črticami, oddaljenost pozicij pa je podana s številko ob črti s krožci. O geometrijsko pravilnem zapisovanju



Slika 2.22: Kartica s štirimi moduli in petimi pozicijami za module.

razdalj se bomo pogovarjali kasneje. Iz podatkov na 2.22 izhajata matriki C in D

$$C = \begin{bmatrix} 0 & 3 & 2 & 4 & 0 \\ 3 & 0 & 4 & 0 & 0 \\ 2 & 4 & 0 & 3 & 0 \\ 4 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 0 & 2 & 5 & 3 & 1 \\ 2 & 0 & 2 & 2 & 3 \\ 5 & 2 & 0 & 3 & 6 \\ 3 & 2 & 3 & 0 & 1 \\ 1 & 3 & 6 & 1 & 0 \end{bmatrix}. \tag{2.22}$$

Vidimo, da sta matriki simetrični in da imata na osnovni diagonali vrednosti 0. Matrika C je sicer reda 4×5, vendar je z ničlami, v zadnji vrstici in zadnjem stolpcu, dopolnjena v red 5×5, ki ga ima tudi matrika distanc D.

Do SMTDP pridemo, če vse elemente, ki so nad osnovno diagonalo matrik C in D uredimo po velikostnem redu v vektor. Vrednosti vektorja $\mathbf{d}$ rastejo v eno smer in vrednosti vektorja $\mathbf{c}$ v drugo smer. Nad obema omenjenima vektorjema priredimo skalarni produkt. Za podani matriki imamo

$$\begin{aligned}\mathbf{c} &= (0,0,0,0,0,2,3,3,4,4)\\ \mathbf{d} &= (6,5,3,3,3,2,2,2,1,1)\\ \mathbf{c}\cdot\mathbf{d} &= 24.\end{aligned} \tag{2.23}$$

Z nasprotnostjo vektorjev zagotovimo, da največji razdalji $\mathbf{d}$ pripada najmanjše število povezav $\mathbf{c}$, kar vodi v optimalno totalno dolžino, ki meri v primeru (2.23) 24 enot.

Če hočemo priti do SMTDPP, moramo postopek ustrezno razširiti. Medtem ko v primeru SMTDP matričnih operacij nad matrikama C in D niti ne izvajamo, moramo to storiti, če le hočemo upoštevati namestitev modulov, permitacijo. Izračunati je potrebno permutacijsko matriko A in sicer tako, da vzamemo i-to vrstico matrike C in j-ti stolpec matrike D ter iz teh matrik izločimo elemente izven osnovne diagonale. Opazovane elemente obeh matrik uredimo v smislu najmanjšega skalarnega produkta, kot smo to napravili pri SMTDP. Tako pridemo do elementa a_{ij} matrike A. Ta matrika tako predstavlja vse možnosti, da se poljuben modul i nahaja na poljubni poziciji (j). Jasno je, da če se nahaja modul i na poziciji (j), se na drugih pozicijah lahko nahajajo le drugi moduli. Za kartico na sliki 2.22 imamo tako matriko permutacij

$$A = \begin{bmatrix} 16 & 18 & 27 & 16 & 13\\ 10 & 14 & 17 & 10 & 7\\ 16 & 18 & 27 & 16 & 13\\ 10 & 14 & 17 & 10 & 10\\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.24}$$

SMTDPP dobimo pri tisti permutaciji, ki z najmanjšo vsoto prekrije vse stolpce matrike A. Torej minimizirati moramo stroškovno (objektivno) funkcijo

$$f(c,d) = 1/2 \sum_{j=1}^{N} a_{ij} \tag{2.25}$$

pri tem, da lahko v vsaki vrstici izberemo le en element a_{ij}.

Postavimo, da imamo normirano vrednost stroškov $f = 0$ tedaj, ko imamo optimalno namestitev modulov. Če nimamo optimalne namestitve,

se ta izkaže z vrednostjo $f = 0 + s$. s so stroški, ki jih doprinaša prav neoptimalnost postavitve. Če najdemo v matriki A N "neodvisnih ničel 0", ki prekrijejo vse stolpce matrike $A(^*)$, pridemo do optimalne rešitve, t. j. SMTDPP s funkcijsko vrednostjo $f = 0$.

In kaj je matrika $A(^*)$? Ta se nanaša na vnašanje neodvisnih ničel po Munkresu [4]. Ta vnašalni postopek ničel zahteva naslednje korake:

1. Za vsako vrstico matrike A določimo najmanjši element a_{min}.

2. Element a_{min} odštejemo od vseh ostalih elementov v opazovani vrstici. Uveljavitev tega pri vsaki vrstice nam da matriko A^*.

3. V dobljeni matriki ponovimo točki 1. in 2., vendar v vertikalni smeri. Kot rezultat dobimo matriko A^{**}, ki ima v vsakem stolpcu in vsaki vrstici vsaj eno ničlo.

4. V matriki A^{**} poiščemo množico z imeni vseh tistih vrstic in stolpcev, ki pokrijejo vse ničle matrike. Naj je ta množica X. Vzemimo, da vsebuje n elementov. Če je $n = N$, je pred nami že optimalna permutacija. Če pa je $n < N$, gremo v naslednjo točko.

5. V opazovani matriki A^{**} izločimo vrstice in stolpce z imeni v množici X in poiščemo v preostanku matrike najmanjši element. Tega prištejemo k vsakemu elementu stolpcev in vrstic, ki imajo svoje ime v množici X. Prištevanje vršimo ločeno za stolpce posebej in vrstice posebej. Za tem omenjeni element odštejemo od vseh elementov matrike A^{**}, kar da matriko A^{***}.

6. Točki 4. in 5. ponavljamo toliko časa, dokler ne pridemo do enakosti $n = N$, kar določa optimalno permutacijo glede na obravnavane stroške. Pri ponavljanju se nam število zvezdic povečuje A^{****}, A^{*****}, A^{******} itd.

Gornji postopek je opisan dokaj na kratko. Zato napravimo zanj z orodjem MATLAB program SMTDP.M, ki se nahaja v Dodatku učbenika. Ta zajema poleg SMTDP na permutacijo tudi splošno dolžino SMTDP. Rezultat, ki ga dobimo z izvajanjem tega programa, je:

```
Connection and Distance Data
c =
0 3 2 4 0
```

```
3 0 4 0 0
2 4 0 3 0
4 0 3 0 0
0 0 0 0 0
d =
0 2 5 3 1
2 0 2 2 3
5 2 0 3 6
3 2 3 0 1
1 3 6 1 0
Lower Bound on Minimal Connection Length: SMTDP
smtd =
24
Permutation Matrices, Minimal Costs Calculation
a =
16 18 27 16 13
10 14 17 10 7
16 18 27 16 13
10 14 17 10 7
0  0  0  0  0
a1 =
3 5 14 3 0
3 7 10 3 0
3 5 14 3 0
3 7 10 3 0
0 0 0  0 0
a11 =
3 5 14 3 0
3 7 10 3 0
3 5 14 3 0
3 7 10 3 0
0 0 0  0 0
a111 =
0 2 11 0 0
0 4 7  0 0
0 2 11 0 0
0 4 7  0 0
0 0 0  0 3
a1111 =
```

```
0 0 9 0 0
0 2 5 0 0
0 0 9 0 0
0 2 5 0 0
2 0 0 2 5
a11111 =
[]
Results with Minimal Costs: Permutations per Element
ep =
1 2 0 4 5
1 0 0 4 5
1 2 0 4 5
1 0 0 4 5
0 2 3 0 0
Results with Minimal Costs: Number of SMTDPP
st =
288
poz=[1 4;2 1;3 2;4 5;5 3]
poz =
1 4
2 1
3 2
4 5
5 3
mincos(poz,a)
Minimum of Cost Function of SMTDPP
ans =
25.5000
```

Matrika `poz` je optimalna namestitvena tabela 2.2. To dobimo na osnovi matrike `a1111` $= A^{****}$, ki vsebuje vse potrebne neodvisne ničle. Do vseh 5 neodvisnih ničel pridemo postopoma iz permutacijske matrike `a` $= A$ tako, da izvajamo Mankresov postopek po vrsti na matrikah `a1`, `a11`, `a111`, `a1111` in `a11111`. Zadnja matrika je prazna (`[]`), kar pomeni, da smo dosegli 5 neodvisnih ničel, ki prekrivajo vseh 5 stolpcev matrike A že z matriko `a1111` $= A^{****}$. Ta matrika zapisana v normalni matematični obliki je

Modul	1	2	3	4	5
Pozicija	(4)	(1)	(2)	(5)	(3)

Tabela 2.2: Ena od optimalnih postavitev modulov na kartico.

$$A^{****} = \begin{bmatrix} 0 & 0 & 9 & \mathbf{0} & 0 \\ \mathbf{0} & 2 & 5 & 0 & 0 \\ 0 & \mathbf{0} & 9 & 0 & 0 \\ 0 & 2 & 5 & 0 & \mathbf{0} \\ 2 & 0 & \mathbf{0} & 2 & 5 \end{bmatrix}.$$

V njej vidimo krepko zapisane ničle, ki so neposredno izbrane za postavitev tabele 2.2. Skupnost teh ničel predstavlja le eno od optimalnih rešitev s stroški $f = 0$. V matriki A^{****} vidimo, da, če hočemo priti do optimalnih stroškov $f = 0$, mora biti modul 5 vedno na poziciji (3), ker je v stolpcu 3 samo ena ničla. Pri drugih modulih obstaja večja ali manjša izbira.

Realno gledano modula 5 sploh nimamo, ker je prvotna pozicija (5) na sliki 2.22 prazna; vse povezave do modula 5 so zato enake 0.

Procesiranje programa SMTDP.M nam da število vseh možnih optimalnih rešitev `st` = 288. Po izrazu (2.19) je vseh postavitev $P = 720$, optimalnih je torej kar precejšnje število. Funkcija `mincos( )` v programu SMTDP.M nam na osnovi matrik `poz` in `a` izračuna SMTDPP, ta meja znaša 25.5 dolžinskih enot, medtem ko je SMTDP = `smtd` = 24.

Če namestimo module po namestitveni tabeli 2.2, dobimo resnično TDP v tabeli 2.3, ki je različna od SMTDPP. Zanimiva je še primerjava s TDP za začetno postavitev modulov na sliki 2.22, ki je postavljena povsem slučajno. Na desni strani tabele 2.3 vidimo, da je ta $\mathrm{TDP}_0 = 45$ dolžinskih enot. Tako pridemo do relacije

$$\mathrm{SMTDP} < \mathrm{SMTDPP} < \mathrm{TDP} < \mathrm{TDP_o}$$

$$24 < 25.5 < 33 < 45\,.$$

Na osnovi te relacije se nam vsiljuje zaključek, da je spodnja meja SMTDPP = 25.5 boljša ocena za TDP kot splošna SMTDP = 24. Z optimizacijo smo pridobili $\mathrm{TDP}_0 - \mathrm{TDP} = 45 - 33 = 12$ dolžinskih enot. Razlika $\mathrm{TDP} - \mathrm{SMTDPP} = 33 - 25,5 = 7.5$ je znatna, zato se nam je splačalo delati optimizacijo. Če bi slučajna začetna postavitev dala TDP, ki bi bila le malo večja ali enaka SMTDPP, optimizacije ne bi delali.

i	j	$c_{ij}d_{ij}$	prod(c,d)	$c_{ij}d_{ij}$	prod$_0(c,d)$
1	2	3 3	9	3 2	6
1	3	2 3	6	2 5	10
1	4	4 1	4	4 3	12
1	5	0 5	0	0 2	0
2	3	4 2	8	4 2	8
2	4	0 1	0	0 2	0
2	5	0 5	0	0 3	0
3	4	3 2	6	3 3	9
3	5	0 2	0	0 6	0
4	5	0 6	0	0 1	0
TDP		Skupaj	33	Skupaj	45

Tabela 2.3: Resnična TDP optimalne in začetne postavitve.

2.5.3 Namestitev modulov na osnovi geometrije Manhattan

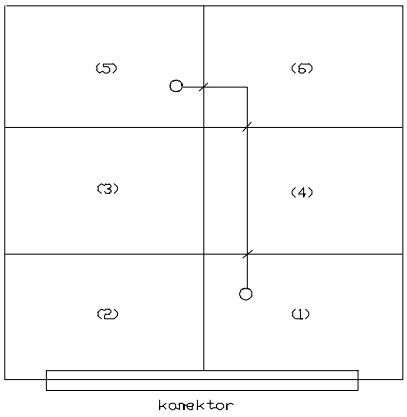
Vzemimo koordinatni sistem in vanj postavimo točki $T_1(x_1, y_1)$ in $T_2(x_2, y_2)$. Razdaljo med omenjenima točkama določimo na osnovi Pitagorovega izreka

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2},$$

ki sodi v okvir elementarne (evklidske) geometrije. Lobačevski je pokazal, da lahko obstajajo poleg splošno znane elementarne geometrije še druge (neevklidske) geometrije. Evklidska geometrija nam največkrat ne daje nobene prednosti pred geometrijo, ki predpostavlja nižjo normo v prostoru. Izraz za D je namreč za množično procesiranje (tega opravlja računalniško orodje za načrtovanje) neugoden, ker zahteva pri vsakem osnovnem računanju odštevanje, seštevanje, množenje in korenjenje, kar zahteva relativno velik procesorski čas. Nižjo normo v prostoru ima geometrija Manhattan, ki upošteva namesto gornje evklidske razdalje D manhattan-razdaljo

$$d = |x_2 - x_1| + |y_2 - y_1|. \tag{2.26}$$

Ta razdalja oziroma ustrezna geometrija Manhattan je uporabna le, če nameščamo module na kartice, ki so koordinatno razdeljene v posamezne pozicije, koordinatna mreža pa je seveda lahko gostejša od gostote pozicij. Tako pripravljeno kartico za 6 pozicij vidimo na sliki 2.23. Na katerokli pozicijo lahko namestimo katerikoli modul. Module lahko povezujemo med seboj samo vodoravno in navpično tako, kot to zahteva mreža. Ena enota



Slika 2.23: Najbolj groba koordinatna mreža za šest pozicij na kartici s prikazom Manhattan-razdalje 3.

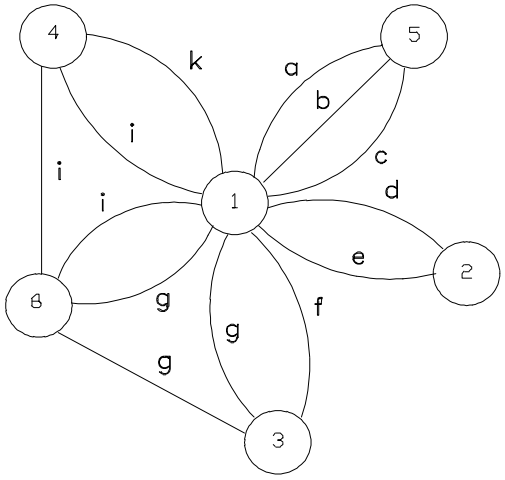
Modul (pozicija)	1 (1)	2 (2)	3 (3)	4 (4)	5 (5)	6 (6)
Dol. povezav	**20**	2	6	3	9	6

Tabela 2.4: Namestitev s TDP = 20.

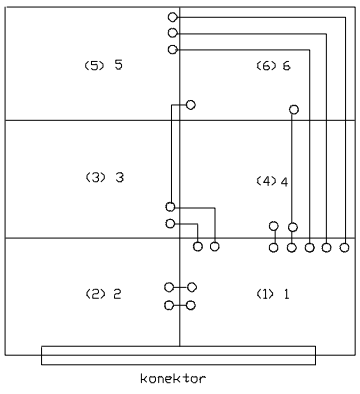
dolžine je en prehod prek meje dveh sosednih pozicij (enkratni prehod preko vodoravne ali navpične linije mreže). Tako ima vrisana povezava na zadnji sliki dolžino 3 enote.

Vzemimo graf RSS na sliki 2.24, ki obsega 6 vrhov in 9 povezav. Kot vemo, je vrh v grafu modul, povezava v grafu pa povezava med ustreznima moduloma. Postavimo graf na kartico na sliki 2.23 tako, da damo modul i na pozicijo (i). Tako pridemo do postavitve na sliki 2.25. V to rešitev so vnešene vse, glede na graf pomembne, Manhattan-razdalje. Prepričamo se lahko, da veljajo dolžine povezav za $i(i)$, ki so podane v razpredelnici 2.4.

V omenjeni razpredelnici vidimo, da je najzahtevnejši modul 1 z dolžino povezav 20 in s SMTDP = 11. Razlika $20 - 11 = 9$ je velika, zato ni dobro, da ostane modul 1 na izbrani poziciji (1). Instinkt nam veleva, da če modul z največjim številom povezav (npr. procesor, kontroler itd., torej čip z velikim številom priključkov) namestimo na sredino (v center) kartice, je pričakovati, da bo TDP manjša. Kako računamo z omenjenimi premikanji modulov? Koliko lahko pridobimo na TDP modulov?



Slika 2.24: Graf RSS s šestimi moduli in devetimi povezavami med njimi.



Slika 2.25: Namestitev s totalno dolžino povezav 20.

$1 \rightarrow i$	2	3	4	5	6
Razlika dol. pov.	-3	**-6**	-4	0	-2

Tabela 2.5: Zamenjave modula 1 z ostalimi moduli.

$3 \rightarrow i$	1	2	4	5	6
Razlika dol. pov.	+6	+1	+1	0	0

Tabela 2.6: Zamenjava modula 3 z ostalimi moduli.

Vzemimo najzahtevnejši modul in ga poskušajmo zamenjati, postopoma, z vsakim od ostalih modulov. Za sliko 2.25 imamo zamenjave $1 \rightarrow i$ v tabeli 2.5 . Vidimo, da nam največji prihranek na dolžini povezav napravi zamenjava $1 \rightarrow 3$, zato to zamenjavo privzamemo.

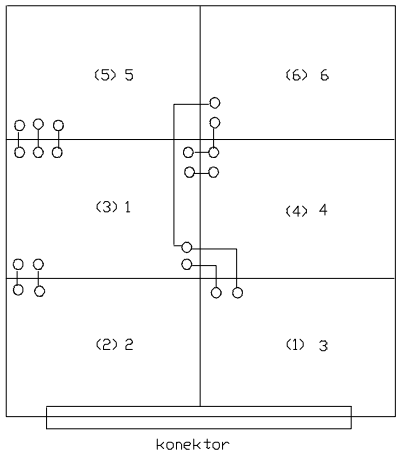
Sedaj imamo modul 1 na poziciji (3) in modul 3 na poziciji (1). Poglejmo, če nam nadaljnje premikanje modula 3 iz pozicije (1) omogoča kakšno izboljšavo. Uporabimo enak postopek za premikanje modulov kot zgoraj, kar nam da novo tabelo 2.6. Iz pozitivnega predznaka in ničel v drugi vrstici izhaja, da z omenjenim premikanjem ne pridobimo nič, kvečjemu lahko kaj izgubimo.

Kakšna je totalna dolžina povezav po zamenjavi modula 1 z modulom 3? Najboljše je, da si narišemo kartico z ustrezno realizacijo. Na sliki 2.26 vidimo, da je vseh prehodov preko mreže 14, kar pomeni, da je TDP = 14.

Postopek optimizacije pa s postavitvijo na sliki 2.26 še ni končan. Zamenjavali smo $1 \rightarrow i$ in $3 \rightarrow i$, s tem pa se nismo dotaknili zamenjav, v katerih ne nastopata modula 1 in 3 (neodvisni moduli). Tudi te zamenjave lahko kaj doprinesejo. Oglejmo si na primer par modulov (4, 6). Modul 4 je neodvisen od modula 3, zato se lahko nahaja, glede na modul 3, kjerkoli. Z modulom 1 ima modul 4 dve povezavi. Modul 6 ima z modulom 1 dve povezavi (kot modul 4), z modulom 3 pa eno povezavo. Če zamenjamo modula 4 in 6, dobimo razpredelnico 2.7, iz katere se vidi, da če premaknemo modul 6 na pozicijo (4) in modul 4 na pozicijo (6) prihranimo 1 enoto dolžine, ker je modul 4 bližje modulu 3. Postavitev na sliki 2.27 upošteva tudi zamenjavo $4 \rightarrow 6$. Vidimo, da ima postavitev TDP = 13. Primerjava

$4 \rightarrow 6$	1 (3)	2 (2)	3 (1)	4 (6)	5 (5)	6 (4)
Dol. povezav.	13	2	4	4	3	4

Tabela 2.7: Učinek zamenjave modulov 4 in 6.



Slika 2.26: Namestitev s totalno dolžino povezav po zamenjavi modulov 1 in 3, TDP=14.

postavitev na slikah 2.25 in 2.27 kaže na to, da je prihranek znaten, saj smo iz TDP = 20 prišli do TDP = 13.

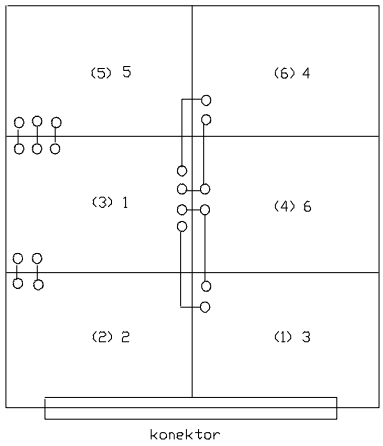
Gornji postopek je preprost, pri realnih RSS pa nekoliko zamuden, saj moramo pregledovati ogromno število zamenjav po dva modula. Glede na to je bolj primeren za računalniška orodja kot pa za praktično delo brez teh orodij. Ali lahko zmanjšamo veliko število parnih zamenjav modulov?

Vzemimo, da sta modula i in j med seboj nepovezana. Če je temu tako, je par (i, j) par tujih modulov. Na osnovi takšnih parov pridemo do množice tujih modulov. Zanimiva je množica, ki vsebuje kar največ parno gledano tujih modulov. Če vzamemo iz te množice katerikoli par (i, j), sta si i in j tuja modula. Na takšni množici je L. Steinberg [5] izdelal naslednji postopek optimalnega nameščanja modulov na kartico:

1. Za neko začetno namestitev modulov poiščemo največjo množico tujih modulov F.

2. Iz kartice z začetno namestitvijo modulov odstranimo vse tiste module, ki niso v množici F. Če je v tej množici t tujih modulov, nam ostane na kartici $N - M + t$ prostih pozicij.

3. Katerikoli modul iz množice F poskušamo namestiti na katerokoli prosto pozicijo. Ker tak modul ni povezan z nobenim drugim mod-



Slika 2.27: Namestitev grafa na sliki 2.24 na kartico s TDP = 13.

ulom v F, lahko računamo dolžino povezav, ne da bi nameščali tudi stalne module na kartico.

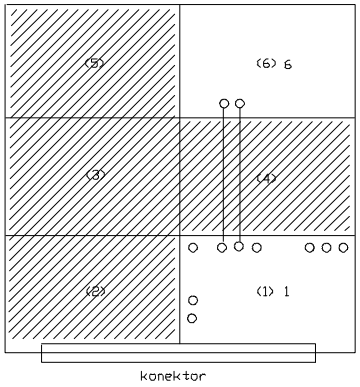
4. Točka 3. nam da pri $N - M + t$ prostih pozicijah, matriko T reda $t\,(N - M - t)$, v kateri so elementi t_{ij} dolžine povezav med moduli iz F če modul i postavimo na pozicijo (j).

5. Na osnovi množice T določimo takšno namestitev t modulov, da je totalna dolžina povezav najmanjša. Totalna dolžina povezav se lahko spreminja le na osnovi modulov, ki so v F, saj je ostali del dolžine povezav konstanten.

6. Če izberemo drugačno začetno namestitev, dobimo lahko drugačno optimalno namestitev. Če gremo preko vseh možnih začetnih na mestitev, nam ne more dati nobena rešitev namestitve boljši rezultat od SMTDP.

Za razlago zadnjega postopka vzemimo začetno namestitev na sliki 2.28. Posamezni koraki postopka so naslednji:
Množico tujih modulov dobimo na osnovi vseh povezanoti (pov) in nepovezanosti (nepov):

(1, 2): pov (2, 3): nepov (3, 4): nepov (4, 5): nepov (5, 6):nepov
(1, 3): pov (2, 4): nepov (3, 5): nepov (4, 6): pov



Slika 2.28: Stalni in gibljivi del modulov za začetno postavitev na sliki 2.25.

(1, 4): pov (2, 5): nepov (3, 6): pov
(1, 5): pov (2, 6): nepov
(1, 6): pov.

Največjo nepovezanost dobimo na osnovi tranzitivnosti $(i, j), (i, k) \rightarrow (i, k)$. Ustrezna množica nepovezanosti je $F = \{2, 3, 4, 5\}$. Pri podatkih $t = |F| = 4, N = 6, M = 6$ imamo $N - M + t = 4$ proste pozicije z matriko T

$$T = \begin{bmatrix} 2 & 4 & 2 & 6 \\ 4 & 3 & 2 & 4 \\ 2 & 4 & 2 & 4 \\ 3 & 6 & 3 & 9 \end{bmatrix}.$$

Vrstice so moduli i in stolpci pozicije (j). Pri postopku optimizacije lahko damo na vsako pozicijo katerikoli modul iz F. Na sliki 2.28 je narisan stalni del (modula 1, 6) in gibljivi del (moduli 2, 3, 4, 5). Stalni del doprinaša dolžino 4, ostali del TDP pa doprinaša gibljivi del.

Za postopek optimizacije pritegnimo MATLAB-program MATT.M, katerega lahko poiščemo v Dodatku. Nanaša se na funcijo `nexta( )`, ki zna določiti matriko z potrebnimi neodvisnimi ničlami. Izvajanje omenjenega programa nam da pri matriki T naslednji rezultat:

```
tma =
2 4 2 6
```

Modul	2	3	4	5
Pozicija	(2)	(3)	5	4

Tabela 2.8: Ena od rešitev s stroški $f = 0$.

```
4 3 2 4
2 4 2 4
3 6 3 9
a1 =
0 2 0 4
2 1 0 2
0 2 0 2
0 3 0 6
a11 =
0 1 0 2
2 0 0 0
0 1 0 0
0 2 0 4
a111 =
[]
0 1 0 2
2 0 0 0
0 2 0 0
0 2 0 4
```

Matrika `a` je, kot rečeno, matrika T, `a1` prva iteracija, `a11` druga iteracija, `a111` pa matrika, ki ni več možna, ker nam že matrika `a11` ponuja vse potrebne neodvisne ničle. V matriki

$$T^{**} = \begin{bmatrix} \underline{\mathbf{0}} & 1 & 0 & 2 \\ 2 & \underline{\mathbf{0}} & 0 & 0 \\ 0 & 2 & 0 & \underline{\mathbf{0}} \\ 0 & 2 & \underline{\mathbf{0}} & 4 \end{bmatrix}$$

so krepko postavljene neodvisne ničle za izbrano optimalno rešitev. V razpredelnici vidimo, da je glede na začetno postavitev sprememba le na pozicijah (4) in (5). Modul 4 je prestavljen na pozicijo (5) in modul 5 na pozicijo (4). Če napravimo to postavitev, imamo TDP = 17. Zadnje je manj od TDP = 20 pri začetni postavitvi in več kot pri optimalni postavitvi

s TDP = 13. Zadnja razlika je nastala zato, ker matrika T predvideva stalni del, ki vsebuje pozicijo (1) in modul 1, ki sta pa pri optimalni rešitvi z TDP = 13 v igri optimizacije (zamenjava modula 1 z modulom 3). Steinbergova rešitev predvideva optimizacijo samo nad gibljivim delom, zato je njen optimum manj učinkovit, kot pri osnovni Munkresovi optimalni postavitvi, ki lahko premika vse module, če je to potrebno. Seveda pa bi prišlo do drugačne optimalne rešitve, če bi delno postavitev 1(3) in 3(1) vgradili v začetno pozicijo, iz katere bi izhajala Steinbergova optimizacija. Prednost Steinbergove optimizacije je v tem, da je matrika T manjša od matrik C in D in da ne opazuje vseh parov $i(j)$, ki jih je v primeru C, D veliko. Slabost pa je v tem, da lahko dobimo pri različnih začetnih nastavitvah različne rezultate. Rezultat pri enem začetnem pogoju je z vidika TDP lahko ugodnejši od rezultata pri drugačnem začetnem pogoju.

2.6 POSTAVLJANJE POVEZAV NA KARTICO

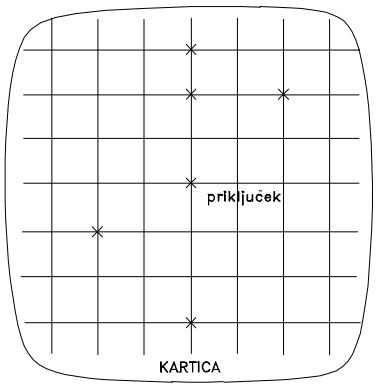
2.6.1 Postavljanje ene povezave

Za veliko RSS smo do sedaj spoznali, kako jo razdelimo na module in kako te module namestimo na kartico. Pred nami je torej kartica z optimalno nameščenimi moduli, ki pa še niso povezani med seboj tako, kot to velevajo medmodulske povezave. V tem podpoglavju si poglejmo, kakšni problemi nastopajo pri fizičnem postavljanju medmodulskih povezav in kako te probleme matematično zajamemo in rešujemo.

Pri postavljanju povezav med moduli (routing) moramo postaviti določene pogoje oziroma primerno okolje, sicer ne moremo tudi teoretično pristopiti k problemu postavljanja povezave oziroma zveze med dvema priključkoma. Sodobna tehnika tiskanin in ožičevanj zahteva naslednje dogovore:

1. Moduli oziroma elektronski elementi se med seboj lahko povezujejo le vodoravno in navpično, kar pogojuje koordinatna mreža, ki jo a priori polagamo (namišljeno) na kartico. Ni nujno, da je mreža pravokotna. Dopustna je krožnost, ovalnost, sferičnost itd., vendar po teh posegamo le, če je ustrezna geometrija na mestu. Največkat ostajamo pri kvadratičnosti ali pravokotnosti, kar omogoča uporabo geometrije Manhattan.

2. Ne sme biti križanja povezav oziroma poti na isti površini kartice. Veliko število križanj zahteva, da kartice izvedemo v več plasteh oziroma



Slika 2.29: Priključki na kartici, ki morajo biti povezani med seboj.

kartica ima svoje strani. Zahtevne RSS (veliki procesorji IBM) zahtevajo več deset strani na kartico. 16-bitna mikroračunalniška tehnika zahteva vsaj šeststranske kartice, 8-bitna tehnika zahteva dvostranske ali štiristranske kartice, itd. Določene strani kartice imajo lahko povsem tehnološko-elektronsko značilnost, kar pomeni, da se z njimi rešuje večja hitrost delovanja, boljšo vzemljitev, omogočajo velike tokovne pretoke itd., ne pa logičnost RSS.

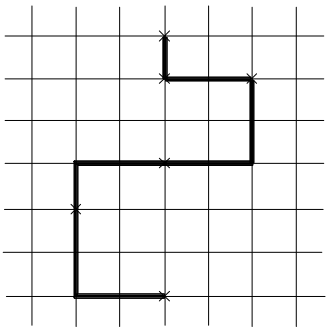
3. Totalna dolžina vseh povezav na kartici mora biti čim krajša, po možnosti minimalna.

4. Število povezav na en priključek je omejeno.

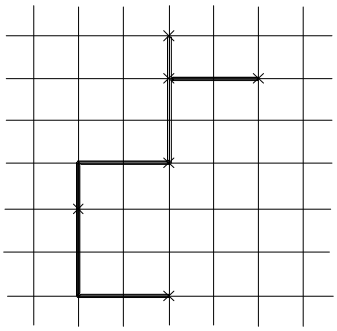
5. Zaradi omejitev povezav na en priključek je možno vpeljati pomožne priključke, pomožne povezave (auxiliary connections), skoznike itd.

V praksi so se uveljavili povsem določeni načini postavljanja povezav. Poskusimo jih obrazložiti na delu kartice na sliki 2.29. Točke $\times$ so priključki, ki morajo biti med seboj povezani pri omejitvi največ dve povezavi na priključek.

Na sliki 2.30 vidimo verižno polaganje povezave oziroma poti (chain routing), ki sodi med najenostavnejše polaganje na kartico. Verižno polaganje na sliki 2.30 zahteva totalno dolžino 14 enot. Naslednja možnost je speto drevo (spanning tree) na sliki 2.31. Totalna dolžina povezave je v tem primeru manjša - samo 12 enot, vendar pa se je pri enem priključku



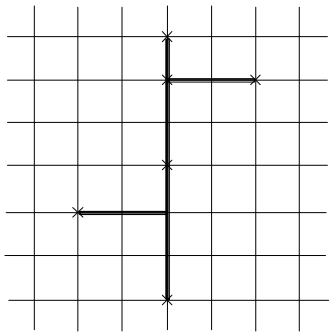
Slika 2.30: Verižna povezava.



Slika 2.31: Speto drevo.

porušil pogoj največ 2 povezavi na priključek. V splošnem je speto drevo učinkovitejše od verižnega polaganja. Možnost na 2.32 predstavlja Steinerjevo drevo. Totalna dolžina povezave je v tem primeru le 10 enot, kar kaže na to, da je Steinerjevo drevo najbolj učinkovito od vseh treh prikazanih. Žrtev za učinkovitost pa je pojav pomožnega, dodatnega priključka (Steinerjeva točka).

Verižno polaganje poti si ne bomo pogledali posebej, ker se v praksi pretežno uporablja bolj učinkovito speto, in še bolj učinkovito, Steinerjevo drevo. Pomembna so takšna speta drevesa, ki imajo najkrajšo povezavo med vsemi vrhovi grafa. Do najkrajše povezave pridemo, če vrh grafa i (to je priključek na kartici) povežemo z najbližjim, sosednim vrhom, recimo,



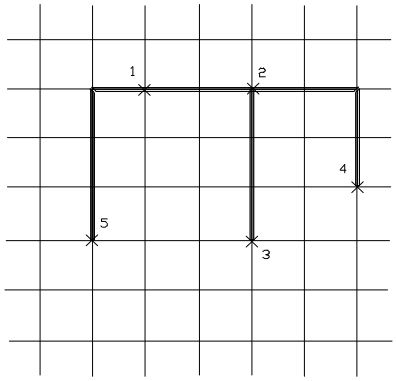
Slika 2.32: Steinerjevo drevo.

da je to j. Razdalja $d(i,j)$ med vrhovoma i in j je $d(i,j) \leqslant d(i,k)$ za vsak vrh k, ki še ni povezan z že povezanima vrhovoma i,j in je najbližji paru i,j.

Naj so na kartici vrhovi grafa (priključki modulov na kartici), ki jih želimo povezovati med seboj. Ustrezna množica vrhov, ki morajo biti povezani med seboj je $V = \{v_1, v_2, ..., v_n\}$. Nadalje vzemimo, da smo že povezali vrhove $v_1, v_2, ..., v_r$. Ti povezani vrhovi predstavljajo fragment vrhov $F \subset V$, $r < n$. K fragmentu F poiščemo najbližjega soseda h kateremkoli vrhu iz fragmenta F. Ta dva povežemo med seboj in na ta način preidemo iz fragmenta z r vrhovi na fragment z $r+1$ vrhovi itd. Če naletimo pri konstruiranju spetega drevesa na to, da imamo več najbližjih sosedov, lahko izberemo kateregakoli od njih, vsak da ustrezno minimalno rešitev. V tem primeru imamo več možnih minimalnih rešitev.

Na osnovi slike 2.33 si oglejmo, kako pridemo do najkrajše poti, ki poveže vse vrhove $v_1, v_2, ..., v_n$ na osnovi operacije min( ). Narisana povezava je slučajna, kar pomeni, da pri njenem postavljanju nismo gledali ali je najkrajša ali pa temu ni tako. Dolžina postavljene povezave je 13 enot. Za dano situacijo priključkov lahko postavimo matriko razdalj

$$D = \begin{bmatrix} 0 & 2 & 5 & 6 & 4 \\ 2 & 0 & 3 & 4 & 6 \\ 5 & 3 & 0 & 3 & 3 \\ 6 & 4 & 3 & 0 & 6 \\ 4 & 6 & 3 & 6 & 0 \end{bmatrix}. \tag{2.27}$$



Slika 2.33: Primer neoptimalnega spetega drevesa.

V tej matriki je zanimiv le trikotnik nad diagonalo. Najmanjši element tega trikotnika je $\min(2, 5, 6, 4, 3, 4, 6, 3, 3, 6) = 2$. Ker ta leži med vrhovoma 1 in 2, moramo ta dva najprej povezati med seboj, kar je že zanetek fragmenta $F = \{1, 2\}$ z $r = 2$. Naslednji korak zahteva najbližji vrh fragmentu. Ta je za $k = 3$, 4, 5

$$\min(d(1,3), d(1,4), d(1,5); d(2,3), d(2,4), d(2,5))$$
$$\min(5, 6, 4, 3, 4, 6) = 3.$$

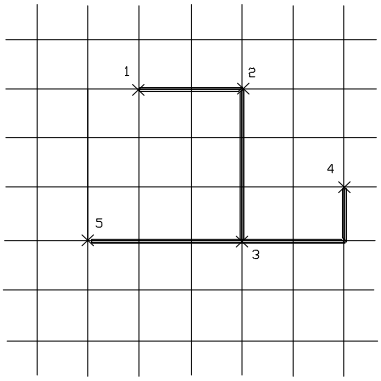
Torej je izbrani $k = 3 = d(2, 3)$. Vrh 3 torej ne povežemo z vrhom 2, temveč z vrhom 3. Ta povezava nas vodi v večji fragment $F = \{1, 2, 3\}, r = 3$. Sledi iteracija

$$\min(d(1,4), d(1,5), d(2,4), d(2,5), d(3,4), d(3,5))$$
$$\min(6, 4, 4, 6, 3, 3) = 3 = d(3,4) = d(3,5)$$

kar pomeni, da kot naslednjo povezavo izvedemo ali 3-4 ali 3-5. Izberemo 3-5. In končna iteracija, manjka nam samo še vrh 4

$$\min(d(1,4), d(2,4), d(3,4), d(5,4)) = \min(6, 4, 3, 6) = 3 = d(3,4).$$

Vrh 4 moramo povezati z vrhom 3, kot nam je nakazala že predhodna iteracija. Rezultat je podan na sliki 2.34. Dolžina povezave je 11 enot,



Slika 2.34: Minimalna povezava vrhov na sliki 2.33.

medtem ko imamo pri slučajni postavitvi na sliki 2.33 dolžino 13 enot. Prihranek je torej za dva prehoda preko mreže (dve dolžinski enoti tipa Manhattan).

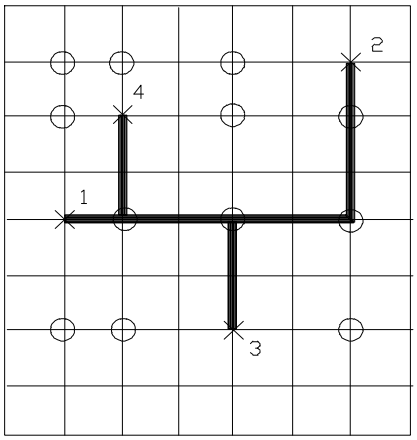
Lahko se dogodi, da je pri primerjavah razdalj $d(i, j)$ najkrajši primerek pri zadnji primerjavi (za čas procesiranja najbolj neugoden primer). Vseh primerjav, ki jih moramo opraviti v tem primeru, je

$$P = \sum_{i=1}^{n-1} i\,(n-i). \tag{2.28}$$

Pri tem je n število vseh vrhov grafa opazovane povezave RSS. V kolikor že imamo fragment z r vrhovi, potrebujemo do konca še naslednje število primerjav

$$P_f = r(r-1). \tag{2.29}$$

Toliko o spetem drevesu. Pojdimo sedaj k Steinerjevemu drevesu. Kot smo že videli, imamo v Steinerjevem drevesu Steinerjeve točke. Vprašanje je, koliko je možnih Steinerjevih točk pri množici vrhov $V = \{v_1, v_2, ..., v_n\}$, ki morajo biti na koordinatni mreži kartice povezani? Skozi vse vrhove položimo vodoravne in navpične vzporednice. Možnost za Steinerjevo točko se pojavi na vsakem preseku vzporednic in navpičnic. Na osnovi slike 2.35



Slika 2.35: Možne Steinerjeve točke ene povezave.

lahko sklepamo, da velja število za izbiro Steinerjevih točk

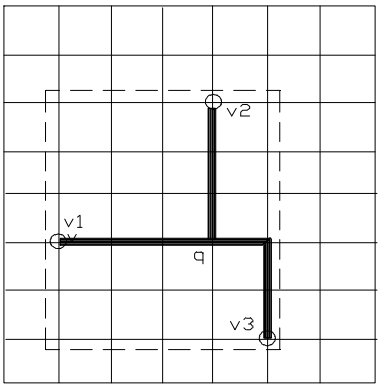
$$S = n^2 - n. \tag{2.30}$$

Pri Steinerjevih drevesih nam je v pomoč najmanjši pravokotnik (trenutno okno procesiranja na opazovani mreži), ki obsega določeno število vrhov grafa. Vzemimo tri vrhove $V = \{v_1, v_2, v_3\}$ na sliki 2.36, ki morajo biti povezani med seboj. V notranjosti najmanšega pravokotnika lahko položimo takšno Steinerjevo točko q, ki minimizira funkcijo

$$f = \sum_{i=1}^{3} d(q,\, v_i). \tag{2.31}$$

Točka q na sliki 2.36 je enako oddaljena od vseh točk v_i in vedno leži v notranjosti najmanjšega pravokotnika (kvadrata), ki zajema vrhove v_1, v_2 in v_3. V notranjost najmanjšega pravokotnika lahko vedno položimo Steinerjevo točko q tako, da je celotna povezava enaka polovici obsega obravnavanega najmanjšega pravokotnika. V gornjem primeru je pol obsega 9 Manhattan-enot.

Do optimalnega Steinerjevega drevesa pridemo na osnovi Hananovega postopka [6], ki glasi približno takole:



Slika 2.36: Najmanjši pravokotnik $P(v_1, v_2, v_3)$.

1. Povezati je potrebno vrhove $V = \{v_1, v_2, ..., v_n\}$. Te položimo v mrežo kot točke v_i s koordinatami (x_i, y_i). Pomembno vlogo igrajo tudi tiste točke, ki sicer niso vrhovi, so pa v najmanjšem obsegu vrhov. Napravimo particijo $P = \{p_0, p_1, ..., p_j\}$ na osi x. V bloku particije p_i so vsi vrhovi nad vrednostjo x_i. Koordinata x je po vrednosti manjša pri p_i in za korak večja pri p_{i+1}.

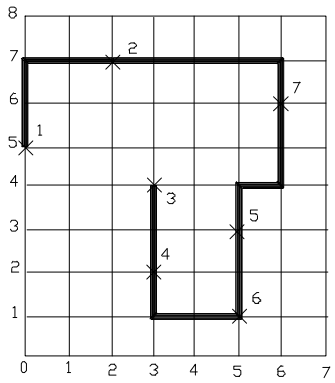
2. Vse vrhove bloka particije p_0, ti so pri x_{min}, povežemo navpično. S tem pridemo do zanetka fagmenta F.

3. Zapišemo fragment F, v katerega damo vse vrhove iz bloka p_i in tudi točke $v_i(x_i, y_i)$, ki ležijo v obsegu vrhov (v notranjosti navpičnice) tega bloka.

4. Izberemo naslednji blok particije P, p_{i+1}, ki je najbližji do p_i, gledano v smeri osi x. V p_{i+1} izberemo točko v_j in v fragmentu F takšno točko f_m, da zadostita pogoju $d(f_m, v_j) \leqslant d(f_q, v_r)$ glede na vse druge točke $f_q(x_q, y_q)$ fragmenta F in vse druge točke $v_r(x_r, y_r)$ v bloku p_{i+1}. $d(f_m, v_j)$ je najmanjša vodoravna razdalja med sosednima fragmentoma.

5. Povežemo točki f_m in v_j in s tem pravzaprav vstavimo vrh v_j v fragment F.

6. Povežemo vse točke v obsegu bloka p_{i+1} in jih tudi damo v fragment F. Tako dobimo novi, večji fragment $F', F \subset F'$.



Slika 2.37: Slučajna verižna povezava sedmih priključkov na koordinatni mreži.

7. Ponavljamo točke 4. do 6. v smislu naraščajočega indeksa blokov i v particiji P.

Pri računanju po Hananovem postopku si lahko pomagamo z MATLAB-programom HPOST.M, ki uporablja funkcijo HANA.M. Oba se nahajata v Dodatku, kjer si jih lahko ogledamo. Za primer uporabe programa in samega postopka vzemimo priključke na sliki 2.37, ki morajo biti povezani z eno povezavo. Na sliki je vrisana verižna povezava, potrebno pa je seveda najti optimalno Steinerjevo povezavo. Iz slike preberemo začetne vhodne podatke, ki so v programu HPOST.M zapisani v obliki matrike `dat1`. Nad temi podatki nam funkcija HANA.M da naslednje rezultate:

```
%PROGRAM TO BE USEFUL BY HANAN's ALGORITHM
%File:HPOST.M
%Create:J.V., LRSS,FRI,Feb.12,1998
%Input data
dat1=[6;3 8;4 3;4 5;6 2;6 4;7 7];
a1=hana(dat1)
dat2=[1 6;2 6;3 6;3 8;4 3;4 5;6 2;6 4;7 7];
a2=hana(dat2)
dat3=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;6 2;6 4;7 7];
a3=hana(dat3)
dat4=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;5 3;6 2;6 4;7 7];
```

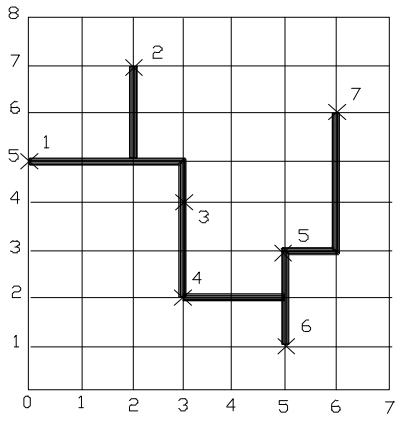
```
a4=hana(dat4)
dat5=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;5 3;6 2;6 4;7,4;7 7];
a5=hana(dat5)
pol1 =
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
a1 =
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
a2 =
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 1 1
0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
a3 =
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 1 1
0 0 1 1 1 1 0 0
0 0 0 0 0 0 0 0
0 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
```

```
a4 =
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 1 1
0 0 1 1 1 1 0 0
0 0 1 0 0 0 0 0
0 1 1 1 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0
a5 =
0 0 0 0 0 1 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 1 1 1
0 0 1 1 1 1 0 0
0 0 1 0 0 0 0 0
0 1 1 1 0 0 0 0
0 0 0 1 1 1 1 0
0 0 0 0 0 0 0 0
```

V programu je `pol1` matrika, ki z elementom 1 predstavlja prisotnost vrha, priključka na kartici. `a1` nam predstavlja particijo danih podatkov v smeri osi y. Funkciji HANA.M sugeriramo enega od najkrajših prehodov v smeri osi x, ko gremo od bloka p_i do bloka p_{i+1} particije P. Možnosti najkrajših povezav v smeri osi x je veliko in sugestija v smislu ene izbrane smeri zelo skrajša čas procesiranja. Ko pridemo iterativno do podatkov `a5`, imamo vse vrhove povezane med seboj, kar predstavlja minimalno Steinerjevo drevo. Na sliki 2.38 imamo izrisan rezultat skladno z matriko `a5`. Na sliki 2.37 imamo dolžino povezave 20, medtem ko je na sliki 2.38 16 enot. Prihranek je torej za 4 prehode preko Manahattan-mreže. Najkrajša pot v smeri osi x dajo točke

(1,6)
(2,6)
(3,6), (3,7), (3,8)
(4,3), (4,4), (4,5), (4,6)
(5,3)
(6,2), (6,3), (6,4)
(7,4), (7,5), (7,6), (7,7).

Opombi: Pri matrikah `pol1` do `a5` sta navpičnost in vodoravnost zamenjani



Slika 2.38: Optimalno Steinerjevo drevo za podatke `dat1`.

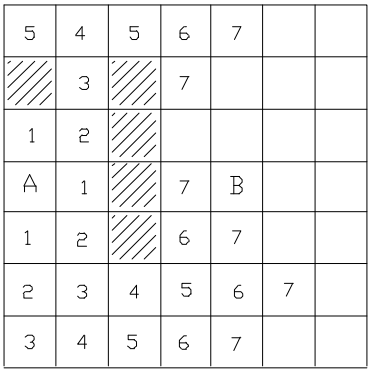
med seboj. Podobno imamo zamenjave tudi v nadaljnjih programih. V programu HPOST.M začenjamo štetje z 1, na sliki 2.38 pa z 0, zato je potrebna v zadnjem spisku točk povsod odšteti 1.

2.6.2 Usklajevanje povezav na kartici

V prejšnjem razdelku smo videli, kako pridemo do ene optimalne poti (povezave), ki jo moramo položiti na kartico. Zadnje bi bilo lepo in prav, v kolikor na kartici ne bi bile predhodno postavljene druge povezave ali kakšne druge ovire, katerim se je potrebno ogniti. Pot lahko položimo, če na trasi ni nobene ovire. Ob pogledu na kartico nam je takoj jasno, da so med posameznimi potmi in drugimi objekti konflikti, zaradi katerih moramo uvajati specifične povezovalne elemente, večstranske kartice (večplastno tiskano vezje), odstopati moramo od optimalnih poti, ki še ne upoštevajo ovire itd. S tem ko položimo i-to pot, napravimo oviro za vse naslednje. Čim več je položenih poti, več ovir obstaja na kartici. Do resnih težav pa pride, ko je gostota povezav na kartici že tolikšna, da naslednje poti ni mogoče ali jih je zelo težko položiti.

Pri polaganju povezav eno za drugo uporabljamo razne variante osnovnega Leejevga postopka [7]. Ta postopek omogoča:

- iskanje optimalne poti med dvema točkama (priključkoma na kartici) ob upoštevanju najmanjšega števila prečkanj z ostalimi, že položenimi



Slika 2.39: Leejevo iskanje poti iz točke A v točko in B.

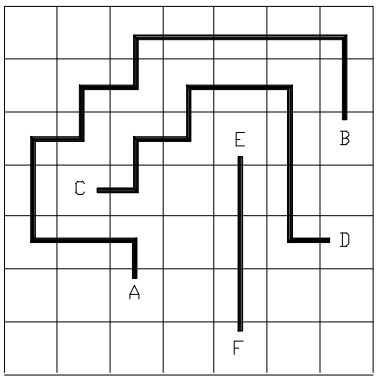
potmi in

- iskanje optimalne poti med dvema točkama ob upoštevanju postavljenih ovir, kot so: že položene povezave, robovi kartic, povezovalni žepi, elektronski elementi, luknje v karticah itd.

Leejev postopek najdemo praktično v vsakem računalniškem orodju, ki se nanaša na polaganje poti (routing), zato si ga moramo ogledati, vsaj v glavnih potezah, tudi v tem delu.

Vzemimo povezovalno situacijo na 2.39. Potrebno je povezati točki A in B tako, da upoštevamo oviri, ki sta podani s šrafirano površino. Najkrajšo pot najdemo po Leeju tako, da preučimo vse možnosti, ki nas vodijo, v našem primeru, iz točke A v točko B. Od začetne točke A začnemo označevati vso mrežo s številkami, ki ustrezajo številu prehodov preko mreže (geometrija Manhattan) gledano od začetne točke. Na sliki 2.39 smo tako že označili celice mreže z 1 do 7, ker nas 7 že privede v ciljno točko B. Dolžina poti, ki leži med A in B je tako 7 Manhattan-enot. Algoritem se ustavi, ko prva številka doseže cilj. Začetna točka je tista točka, ki ima največ ovir, ker je v tem primeru potrebno polagati najmanjše število številk (najmanjši čas procesiranja). Zadnje pa ni pomembno, če opazujemo povezovalni problem le v domeni najmanjšega pravokotnika, ki že vsebuje možno pot med A in B.

Algoritem po Leeju je enostaven, če ga izvajamo s svinčnikom na papirju. Temu pa ni tako v primeru programa, ker je potrebno zadostiti



Slika 2.40: Tri različno dolge povezave na kartici.

vsem možnim oviram na poti med A in B. Vsa stvar se razrašča v smislu razcepljanja in je težko najti kratke univerzalne rutine. Poskus takšnega programa predstavlja MATLAB-program LEEA.M, ki ga najdemo med ostalimi programi v Dodatku. Program potrebuje dve funkciji `plus1( )` in `crosbar( )`, ki se nahajata v Dodatku za omenjenim programom. Prva omogoča postavljanje številk od A naprej in druga, korekcijo polja številk v senci za oviro.

Program LEEA.M da naslednji rezultat

```
mfild =
3 2 1   111 1   222 5
4 3 2   1   2   3   4
5 4 222 222 222 222 5
6 5 6   7   0   7   6
7 6 7   333 0   0   7
0 7 0   0   0   0   0
0 0 0   0   0   0   0.
```

Ta se ujema s sliko 2.39, če vzamemo, da je $A = 111$, ovira 222 ... 222, točka $B = 333$ in neoznačena celica 0. Vidimo, da obstajata dve poti dolžine 7, obe z leve strani druge ovire. Če bi vzeli namesto 333 = (5,4) na primer 333 = (5,5), bi prišlo do dveh poti dolžine 8, ene z leve in ene z desne strani ovire.

Pri sekvenčnem polaganju več poti je važen vrstni red polaganja leteh. Na sliki 2.40 vidimo tri povezave $A-B$, $C-D$ in $E-F$. Značilnost teh

povezav je, da je prva najdaljša, druga srednje dolga in tretja najkraša. Če bi položili povezavo $A-B$ pred $E-F$ in $C-D$, ne bi mogli rešiti problema brez prečkanja poti. Do najmanjšega števila prečkanj pridemo, če položimo vedno najprej najkrajšo pot/povezavo (route shortest wires first). Možen pa je tudi obraten vrstni red, če se držimo tudi ustreznega obratnega algoritma.

Ogledali smo si le nekaj osnov Leejevega algoritma. Če bi se hoteli poglobiti v vse detajle in razne izpeljanke, bi bili mnogo bolj obsežni. V omenjene detajle se morajo spustiti inženirji, katerih delovna opravila temeljno zajemajo problematiko ožičevanja, tiskanih vezij, večstranska tiskana vezja, pisanje CAD programske opreme in druge spremljajoče tehnike, ki so potrebne pri realizaciji sodobnih računalniških sistemov.

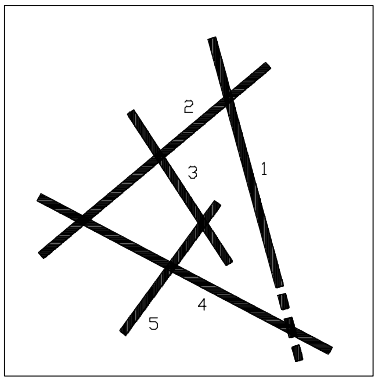
2.6.3 Večstranske kartice

Če hočemo položiti na kartico evklidske povezave, ki jih vidimo na 2.41, je očividno, da tega ne moremo storiti, če imamo na voljo samo enotranske kartice. Problem pa je rešljiv, če lahko posežemo po večstranskih karticah, t. j. po večslojnem tiskanem vezju (multilayer PCB). Koliko strani pa pravzaprav potrebujemo za podano povezovalno specifikacijo (končno množico povezav)? Za povezave na sliki 2.41 (črtkani del povezave 1 naj nas ne moti - vzemimo ga kot nečrtkanega) je problem zanesljivo rešljiv s 5-stransko (5-slojno) kartico, saj v tem primeru damo lahko na vsako stran eno povezavo, ki se ne more sekati z nobeno drugo, ker te druge ni na opazovani strani.

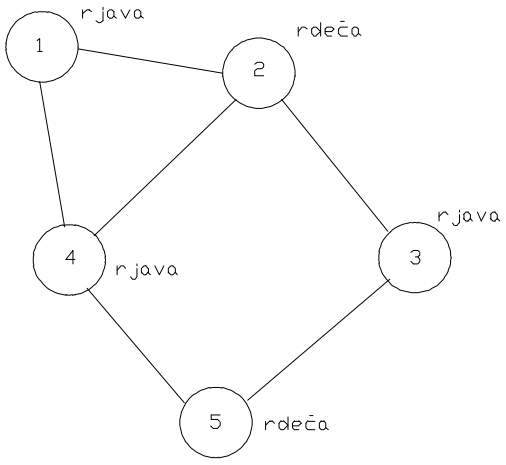
Zaradi tehnoloških, stroškovnih in zanesljivostnih performans želimo vedno položiti vse povezave na kartico z najmanjšim možnim številom strani. Kartico z manj stranmi je namreč lažje in ceneje izdelati kot kartico z več stranmi. Glede na to se vprašamo, če problem na sliki 2.41 ni rešljiv z eno stranjo, ali je ta problem rešljiv na 2-stranski kartici?

Dodelimo vsaki povezavi na sliki 2.41 spojišče v grafu. Dve spojišči i in j povežemo, če se ustrezni povezavi sekata. Tako pridemo do grafa na 2.42. Vidimo, da je dobljeni graf mnogo bolj prijazen od samih povezav, če je le potrebno rešiti problem križanj povezav na kartici.

Kromatično število (chromatic number) določenega grafa je število različnih barv, s katerimi pobarvamo spojišča v grafu tako, da nobena povezava ne povezuje dveh spojišč enake barve. Kromatično število je za tehniko tiskanih vezij pomembna performansa, saj je ta enaka številu potrebnih strani kartice, ki omogočajo, da ne pride do nobenega preseka povezav. Na sliki 2.42 smo že skušali označiti (pobarvati) vrhove tako, da bi vsaki



Slika 2.41: Evklidske povezave na kartici.



Slika 2.42: Graf križanj povezav na sliki 2.41.

povezavi na konceh pripadala različna barva. Vzeli smo rjavo in rdečo barvo. Do spornosti pride le pri vrhu 4, saj bi moral ta vrh imeti rdečo barvo zaradi vrha 1 in hkrati rjavo barvo zaradi vrhov 2 in 5. Povezava med vrhovoma 1 in 4 je neprimerna za kromatsko število 2. Torej križanju evklidskih povezav na sliki 2.41 samo z dvema barvama (beri: z dvostransko kartico) nismo kos.

Koenig [7] je postavil naslednje preprosto pravilo: Vse vrhove grafa lahko pobarvamo z dvema različnim barvama, če nima nobenega cikla s lihim številom vej. Tak cikel se namreč na sliki 2.42 pojavi med vrhovi 4-1-2-4, zato ima ustrezno križanje povezav kromatsko število, ki je večje od 2.

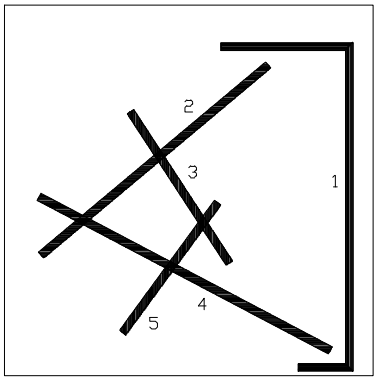
Če na sliki 2.42 skrajšamo povezavo 1 tako, da ne seka povezave 4 (glej črtkani del povezave 1), dobimo nov graf, ki nima povezave med vrhovoma 1 in 4. S tem problem lihega cikla 4-1-2-4 odpade in preostali povezovalni situaciji ustreza kromatsko število 2.

Imenujmo povezavo grafa, ki ima na svojih koncih različno obarvana vrhova - primerna veja. Neprimerna pa je tista veja, ki ima na obeh koncih vrhova z enako barvo. Za praktično delo so zanimivi takšni grafi, ki imajo minimalno število neprimernih povezav. Na sliki 2.42 je na primer 1-4 neprimerna veja, medtem ko je veja 1-2 primerna.

Rez je vsaka množica vej v grafu, ki, če jih odstranimo, razdelijo vse vrhove v dve nepovezani množici. Tak rez je zelo pomemben, če hočemo priti do grafa, ki ima minimalno število neprimernih vej. Graf z dvema barvama je minimalen, če ima vsak njegov rez najmanj polovico primernih vej. Če na sliki 2.42 odstranimo veji v grafu 1-4 in 1-2 (prva je neprimerna in druga primerna), dobimo dva podgrafa, v prvem ni nobene veje, v drugem pa so vse veje primerne.

Kromatična števila niso od muh, saj je v njihovem kontekstu obstajal dolgo časa za matematike dokaj težko rešljiv problem. Formalno je namreč težko dokazati, da so za barvanje kateregakoli ravninskega grafa dovolj le štiri različne barve. Če vzamemo množico povezav na kartici, ki se dajo razgrniti v ravninski graf, se da problem križanj na osnovi gornjih pravil vedno rešiti s 4-stransko kartico. Kot rečeno, formalni dokaz kromatskega števila 4 je težko postaviti sicer pa je očividno, da se da samo s štirimi barvami različno pobarvati na primer vse države tega sveta na istem zemljevidu tako, da so sosedne države različno pobarvane.

Izkustva zadnjih desetletij so pokazala, da so 2-stranske kartice izredno učinkovite, če na eni strani polagamo samo navpične in na drugi strani strani samo vodoravne povezave. Vrsta računalniških pomagal predvideva



Slika 2.43: Reševanje povezovalnega konflikta na osnovi neevklidske povezave.

prav ta osnovni "routing princip" na karticah. Slabost 2-stranskih kartic je v tem, da je potrebno poševno povezavo najprej razdeliti na vodoravno in navpično komponento in se tako ena povezava znajde na obeh straneh 2-stranske kartice. 2-stranske kartice imajo zato veliko število skoznikov in nadaljevanj povezave na drugi strani kartice, kar je nezaželeno. Da bi dosegli učinkovitejše polaganje povezav, se dopušča, da povezave z določenim majhnim naklonom (par stopinj) ne delimo na povezavo na eni in povezavo na drugi strani. Povezava torej ostane cela, na eni ali drugi strani kartice. Na ta način lahko prihranimo nekaj skoznikov, malo nagnjene povezave pa še ne vnašajo neoptimalnosti (gostota na kartici je zadostna).

Problem povezav na sliki 2.41 je takoj rešljiv na 2-stranski kartici, če je povezava 1 neevklidska. Z uvajanjem geometrije Manhattan in Leejevega algoritma, pridemo do slike 2.43, ki nima več opazovanega lihega cikla. Če ta proces umikanja oviram nadaljujemo, pridemo celo do rešitve problema z 1-stransko kartico.

V manjših, tudi mikroračunalniških sistemih največkrat uporabljamo tiskanine na obeh straneh kartice, t.j. 2-stranske kartice. Pri 16-bitni, še bolj pa pri 32-bitni tehniki že uporabljajo 4- in 6-stranske kartice. Če gremo od mikroračunalnikov k velikim računalnikom, se število strani na tiskano kartico povečuje, povečujejo pa se tudi dimenzije kartice. Tako je imel IBM že relativno zgodaj 32-stransko kartico, japonske RSS pa tudi to številko krepko presegajo. Osrednje strani se uporablja za doseganje logičnih, elek-

tronskih in drugačnih vzemljitev. Večkrat so ravno vzemljitveni sistemi tisti, zaradi katerih se odločajo za več stranske kartice.

2.6.4 Programirno postavljanje povezav

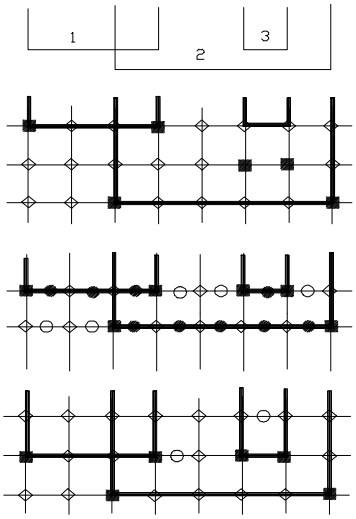
S tem ko so se začeli pojavljati programirni čipi (PLD, FPGA in drugi večji programirni elementi), je fizično načrtovanje kartic nekoliko izgubilo na pomenu, saj se je stopnja integracije vezij (pri teh gre za načrtovanje v notranjosti enega modula, mikroelektrosko načrtovanje) na splošno zelo povečala. Ker imamo v tem času čipe z že desettisoči ekvivalentnih vrat na čip, je jasno, da se je pomen kartic na nivoju osnovnih komponent relativno zmanjšal. Ne smemo pa pozabiti, da je potrebno tudi velike čipe, z velikom številom ekvivalentnih vrat, nameščati in med seboj povezovati, kje drugje kot na kartici. Glede na to bo fizično načrtovanje kartic v klasičnem pomenu besede še dolgo prisotno v računalništvu.

Velike integrirane komponente se po drugi strani vedno bolj programirno odpirajo kasnejšemu načrtovalcu. Kasnejši načrtovalec je tisti, ki načrtuje RSS po že proizvedenih komponentah. Proizvedene komponente niso funkcionalno dokončne, funkcionalno dokončnost vnese šele omenjeni načrtovalec. Programirno odpiranje čipov ima za posledico, da začenja fizično načrtovanje dobivati drugačne in to velike razsežnosti. Programirne možnosti se namreč kasnejšemu načrtovalcu ne odpirajo samo z vidika načrtovanja logike, temveč tudi z vidika povezav med logičnimi podsestavi na čipu.

Za potrebe programiranih čipov so začeli uporabljati tako imenovano segmentno zamotano povezovanje (maze routing), ki v principu izkorišča Leejev algoritem in optimizacije na osnovi geometrije Manhattan. V smislu učinkovite 2-stranske kartice postavljajo navpične povezave na eni strani (vertical tracks) in vodoravne povezave na drugi strani plasti (horizontal tracks). Pred desetletjem so poznali pri 2-stranskih karticah kovinske skoznike, ki so prepeljali povezavo iz ene strani na drugo stran kartice. Z razvojem tehnologije je ta skoznik postal programirni element (fuse, antifuse) v notranjosti čipa, kar pomeni, da ga šele kasnejši načrtovalec RSS v svojem laboratoriju zaprogramira ali pa ga pusti neprogramiranega. Na ta način lahko logični element (lahko tudi večji logični modul), ki je na primer na navpični povezavi p_i, povežemo z vsakim elementom, ki je tudi na navpični povezavi p_j, $j \neq i$. Programirni element v omenjenem smislu vnaša v sodobno fizično načrtovanja RSS povsem nove poglede in tudi nove postopke povezovanja v RSS.

Kakorkoli, programirni element ni več kovinski skoznik, kot je bil to v primeru klasičnih 2-stranskih karticah. Z vidika elektrotehnike je to upor, ki ima lahko veliko ali majhno upornost. Z vidika mikroelektronike je ta isti upor integriran tranzistor oziroma dioda v svojstvu svoje zaprtosti (ON) ali odprtosti (OFF). Osnovni programirni element je "fuse", kar pomeni v žargonu, da, če ga s programiranjem ne spremenimo (unblown), ostane v stanju majhne ohmske vrednosti (idealno 0 ohmov). Če ga zaprogramiramo, mu spremenimo stanje v veliko ohmsko vrednost (idealno ∞ ohmov). "Antifuse" je tehnološko tako postavljen, da se vloga programiranja v smislu majhne in velike ohmske vrednosti obrne.

Pri vgrajenih vodoravnih in navpičnih povezavah v programirnem čipu potrebujemo hkrati fuse- in antifuse-elemente. S fuse-elementi segmentiramo postavljene vodoravne povezave in z antifuse-elementi povezujemo navpično povezavo s segmentom vodoravne povezave, ki leži pod elementom. Segmentiramo vedno le vodoravne povezave, navpičnih povezav ne segmentiramo, saj so na njih logični podsestavi čipa.



Slika 2.44: Princip programirnega povezovanja v programirnih čipih.

Princip programirnega postavljanja povezav vidimo na sliki 2.44. Opraviti imamo s štirimi deli slike. Zgornji del prikazuje šest navpičnic, ki morajo biti paroma med seboj vodoravno povezane tako kot vidimo na

omenjeni sliki. Ustrezne tri povezave so označene z 1, 2 in 3. Drugi del slike prikazuje postavitev povezav na nesegmentni način, tretji prikazuje polno segmentno in četrti 1-segmentno postavitev povezav. Pomen posameznih simbolov na sliki 2.44 je naslednji:

■	zaprogramiran antifuse-element (ohmska upornost 0 ohmov), povezava med navpično in vodoravno linijo
○	zaprogramiran fuse-element (ohmska upornost ∞ ohmov), ločitev dveh sosednih segmentov
◊	neprogramiran antifuse-element (ohmska upornost ∞ ohmov), nepovezava med navpično in vodoravno linijo
•	neprogramiran fuse-element (ohmska upornost 0 ohmov), povezava dveh sosednih segmentov.

Nesegmentna rešitev nima nobenega fuse elementa. Polno segmentna rešitev ima fuse-element dosledno med dvema antifuse-elementoma. V enosegmentni rešitvi povezav ima v vsakem vodoravnem vodilu le en fuse-element. Na sliki 2.44 so povezave 1, 2 in 3 krepko izrisane.

O programirnem postavljanju povezav v okviru IC-vezij bomo v naslednjih poglavij še precej govorili.

2.7 TEHNOLOGIJA TISKANEGA VEZJA

2.7.1 Uvod

V naših laboratorijih večkrat pripravljamo razna tiskana vezja (tiskanine) za RSS, ki jih vstavljamo v PC, delovne postaje in tudi večje računalniške sisteme. Ker so takšna vezja individualna, ne naprej industrijsko pripravljena, je na mestu, da podamo kratek tehnološki pregled izdelave le-teh.

2.7.2 1-plastno tiskano vezje

V tem poglavju smo že obravnavali kartico v RSS, vendar smo to napravili metodološko, brez opisovanja tehnološkega ozadja, ki spremlja kartico. Ta kartica je v osnovi tiskano vezje, na katerega nameščamo komponente. Vezje ima dve osnovni funkciji: omogoča medsebojno električno povezavo med komponentami in daje mehansko oporo (podlago) za komponente na vezju. Pritrditev komponent na tiskano vezje je običajno izvedeno s spajkanjem, lahko pa se pritrdijo tudi s vijačenjem, netanjem itd., možne pa so tudi kombinacije pritrditev. Izbira načina pritrditve zavisi od mehanskih

lastnosti komponent, velikosti tokov, ki tečejo skozi komponente in povezave na tiskanem vezju, od zamenljivosti komponent itd.

Osnova za tiskano vezje je običajno trda plošča iz pertinaksa, vitroplasta ali keramike. Materiali so tako izbrani, da so odlični izolatorji in imajo nizko dielektično konstanto. V novejšem času se najpogosteje uporablja vitroplast, ki ima tudi izredno dobre mehanske lastnosti. Pertinaks ima slabše mehanske lastnosti in se uporablja predvsem zaradi ugodne cene. Keramika se zaradi visoke cene in bolj zapletene tehnologije izdelave uporablja predvsem za visoko frekvenčne aplikacije ter hibridna vezja.

Najenostavnejša so 1-plastna (stranska) tiskana vezja. To pomeni, da so povezave samo na eni strani tiskanega vezja. Osnova za izdelavo tiskanega vezja je osnovna plošča, na katero je nanešena prevodna plast. Ta plast je največkrat bakrena, ker se baker dobro obnese tako z vidika elektrike kot tudi z vidika kemične obstojnosti (kemični procesi staranja v principu ne zmanjšujejo življenske dobe RSS).

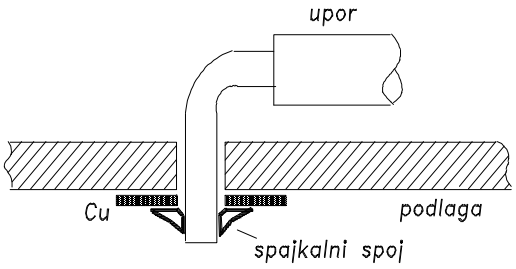
Prvi korak pri izdelavi tiskanega vezja je vrtanje lukenj za komponente oziroma prevezave. V drugem koraku se na prevodno površino s posebno barvo nariše sliko električnih povezav. Običajno se slika električnih povezav nanese s pomočjo sitotiska. Pri zahtevnejših (natančnejših) vezjih pa je potreben foto postopek. Barva oziroma foto lak predstavlja zaščito bakrenih površin. Nezaščitene površine s pomočjo kisline odstranimo tako, da na osnovni plošči ostane prevodna snov (baker) samo tam, kjer je bila barva ali foto lak. Komponente na tiskano vezje pritrdimo tako, da priključne sponke komponent vstavimo skozi pripravljene luknje v tiskanem vezju in jih na nasprotni strani prispajkamo na prevodno površino (kot smo rekli, običajno bakreno). Primer pritrjevanja komponente je v prerezu prikazan na sliki 2.45.

Eno plastna tiskana vezja se uporabljajo v enostavnih računalniških strukturah, v katerih ni veliko povezav. Zanimiva so predvsem zaradi zelo ugodne cene in enostavne izdelave.

2.7.3 2- in večplastna tiskana vezja

Ker so sodobne komponente zelo kompleksne je za njihovo povezavo med seboj potrebnih zelo veliko povezav. Večanje števila medsebojnih povezav zahteva kompleksnejša tiskana vezja. Ker je gostota povezav na takšnem vezju zaradi električnih in mehanskih lastnosti omejena, je potrebno povečati število povezovalnih plasti.

Najenostavnejši način za povečanje povezovalnih plasti je 2-plastno tis-



Slika 2.45: Električna in mehanska postavitev upora na 1-plastno tiskano vezje.

kano vezje, kar pomeni, da so prevodne plasti na strani komponent in strani spajkanja. Ker so povezovalne plasti na obeh straneh, je potrebno omenjeni plasti tudi ustrezno povezati. Za povezovanje plasti je potrebno metalizirati luknje (skozniki), ki povezujejo stran komponent in stran spajkanja. Luknje so lahko izvedene za vstavitev komponent ali samo za električno povezavo plasti. Metalizacija izvrtanih lukenj se izvede s kemijskim nanosom kovine na površino izvrtanih lukenj in v drugem koraku se nanos kovine poveča s pomočjo galvanizacije. Izdelava povezav je enaka, kot pri eno stranskem tiskanem vezju, vendar na obeh straneh.

2-plastna tiskana vezja v modernem računalništvu zajemajo sorazmerno velik delež. Prednosti 2-plastnih tiskanih vezjih je v sorazmerno enostavni izdelavi, kar pomeni tudi nizko proizvodno ceno. Čeprav gre za relativno nizko ceno 2-plastnih vezij, imamo pri pravokotni postavitvi povezav veliko sposobnost postavljanja povezav, kar smo že obravnavali in ugotavljali v razdelku 2.6.3.

Kjer gostota povezav preseže gostoto 2-plastnih tiskanih vezij, so potrebne nove, kompleksnejše rešitve. Povečevanje gostote komponent vodi v večplastna tiskana vezja. Pri teh tiskanih vezjih so plasti s povezavami tudi v samem tiskanem vezju, to je med stranjo komponent in stranjo spajkanja. Grobo si lahko predstavljamo izdelavo večplastnih tiskanih vezij, kot zloženko 1-plastnih. Pri večplastnih vezjih je zelo pomembna natančnost sicer se notranje plasti ne pokrivajo z zunanjimi. Potrebna tehnologija za izdelavo večplastnih tiskanih vezij je neprimerno bolj zahtevna, kot pri 2-plastnih, kar se odraža zlasti v ceni vezja.

2.7.4 Moderna tiskana vezja

Pri modernih tiskanih vezjih niso dovolj le električne povezave in mehanska trdnost. Vezje mora biti tudi ustrezno zaščiteno pred zunanjimi vplivi. Zaščito običajno napravimo s tako imenovanim stop lakom. Ta nanos ima dve nalogi: ščiti tiskano vezje pred atmosferskimi vplivi ter onemogoča spajkanje na površinah, ki niso za to namenjene. To je predvsem pomembno pri strojnem spajkanju.

Za lažje orientacijo (pri proizvodnji, testiranju, popravljanju itd.) je na tiskanem vezju narisan razpored komponent. To je slika postavitve komponent. Ta slika se imenuje beli tisk. Polek razporeda komponent so zapisane tudi drugi podatki, ki olajšajo delo pri opazovanju vezja. Tako spoznamo mesto priključkov, funkcije mikrostikal, mesto vzemljitve itd.

2.7.5 Potrebni podatki za izdelavo tiskanega vezja

Za izdelavo modernega tiskanega vezja je potrebno izdelovalcu posredovati ustrezne podatke, največkrat kar v digitalni obliki. Za izdelavo vezja so tako potrebni naslednji podatki:

1. *digitalna slika povezovalnih plasti.* Pri 2-plastnem tiskanem vezju je to stran komponent in stran spajkanja. Če imamo večplastno tiskano vezje, je potrebo izdelati podatke za vsako plast posebej.

2. *digitalna slika za stop lak.* Za enostavna vezja je slika za stran komponent in stran spajkanja enaka.

3. *digitalna slika za beli tisk.* Pri komleksnih vezjih so komponente na obeh straneh, kar pomeni beli tisk za obe strani.

4. *vrtalna shema.* To so podatki, ki rabijo za avtomatsko strojno vrtanje lukenj.

Omenjene slike vezij se običajno posredujejo izdelovalcu v gerber-formatu. Ime izhaja iz proizvajalca foto risalnika, ki je tudi definiral format zapisa v digitalni obliki. Vrtalna shema je zapis koordinat lukenj za vsako debelino svedra posebej.

Poleg podatkov za izdelavo tiskanih vezij so pri moderni proizvodnji tiskanih vezij potrebni še naslednji podatki:

- *podatki za avtomatsko postavljanje komponent* na tiskano vezje in

- *slika paste za spajkanje.* To je slika točk spajkanja, ki jo potrebuje strojno spajkanje. Zadnje velja za komponente, ki so pritrjene samo na eni strani (SMD - surface mounted components).

Vsi podatki za izdelavo in montažo tiskanega vezja so rezultat modernih orodij za načrtovanje tiskanih vezij (na primer P-Cad). To pomeni, da načrtovalec posreduje vse potrebne podatke v digitalni obliki, izdelovalec pa s pomočjo teh podatkov izdela tiskano vezje.

V ilustracijo podajmo primer podatkov iz laboratorijske prakse LRSS, FRI-Ljubljana, potrebnih za tiskano vezje 'flash' pomnilnika. Izhodiščna datoteka FLASH.PCB je kodirana datoteka, nad katero deluje orodje P-Cad, zato jo eksplicitno brez orodja ne moremo podati. Iz te datoteke izhaja datoteka FLASH.INS, ki predstavlja podatke za CNC (koordinatno numerično kontrolirano) postavitev komponent na tiskano vezje. Ta datoteka izgleda takole

```
%*************************************
%                                    *
% PC-INSERT FILE                     *
%                                    *
% PC-INSERT Version 5.10             *
% Date :  MAY 25 1998                *
% Time :  08:15 AM                   *
% File In :  FLASH.PCB               *
% File Out :  FLASH.INS              *
%                                    *
%*************************************
$FORMAT
 REFDES CENTX CENTY ANGLE TYPE
$UNITS
 MIL
$BOARD-EXTENT
 0 3550 4750 0
$BOARD-ORIGIN
 0 3550
$TARGET
$THROUGH-TOP
 IC8 3900 -300 270 74LS74
 C11 3875 -3340 90 CASE-AA
 C1 1100 -900 90 CK06
```

```
.
.
.
 S3 1875 -775 180 RSIP9
 Q1 4549 -300 90 T237EBC
$THROUGH-BOTTOM
$SMD-TOP
 IC1 1113 -575 180 HC245D
 IC2 1113 -1275 180 HC245D
 IC3 1113 -2275 180 HC245D
 IC4 1113 -2925 180 HC245D
 IC5 2188 -575 180 HC245D
 IC6 2188 -2275 180 HC245D
$SMD-BOTTOM
 A -250 -1775 0 PLCC100
 B -4500 -1975 0 PLCC100
$END
```

Za vsako komponento imamo v datoteki 4 podatke: ime v logični shemi, center v x-smeri, center v y-smeri in ime komponente po katalogu (bazi komponent).

Sledi datoteka FLASH.DRL, na osnove katere poteka avtomatsko vrtanje lukenj na tiskanem vezju

```
/%*************************************
/%                                    *
/% Program :  PC-DRILL VERSION 5.10    *
/% Date :  Apr 14 1998                 *
/% Time :  11:43:48 AM                 *
/% File In :  FLASH.PCB                *
/% File Out :  FLASH.DRL               *
/% Format :  PC-DRILL OUTPUT LIST      *
/% Layer Names :  <ALL>                *
/%                                    *
/%*************************************
/DBGRID 1
/DBUNIT 2
M72
T1C0.700
X003750Y-004750
```

```
X004750Y-004500
X005250Y-005000
X005750Y-005500
X006250Y-006000
X006750Y-006500
X007250Y-007000
.
.
.
X023000Y-026250
X021000Y-026250
X012000Y-032500
X010000Y-032500
X010000Y-026000
X012000Y-026000
T3C1.000
X011000Y-034000
X012000Y-034000
M30
G54D18*X04502Y05712D03*X04502Y05812D03
*X04502Y05912D03
```

V tej datoteki je koordinatni podatek (X, Y) za vsako luknjo in podatki za premer svedra ter drugi podatki za orodje, ki izvaja vrtanje.

Tudi fotoploter zahteva svoje podatke. Začetek in konec teh podatkov (gerber-format) vidimo v datoteki

```
*X04502Y06012D03*X04502Y06112D03
*X04502Y06212D03*X04502Y06312D03
*X04502Y06412D03*X05602Y06412D03
*X05602Y06612D03*X06977Y06387D03
*X06977Y06587D03*X04727Y05887D03
*X04927Y05887D03*X03627Y05887D03
*X03827Y05887D03*X03627Y05187D03
*X03827Y05187D03
.
.
.
*X03727Y03287D01*X06452Y03287
*X06677Y03287*X07327Y06587D02
```

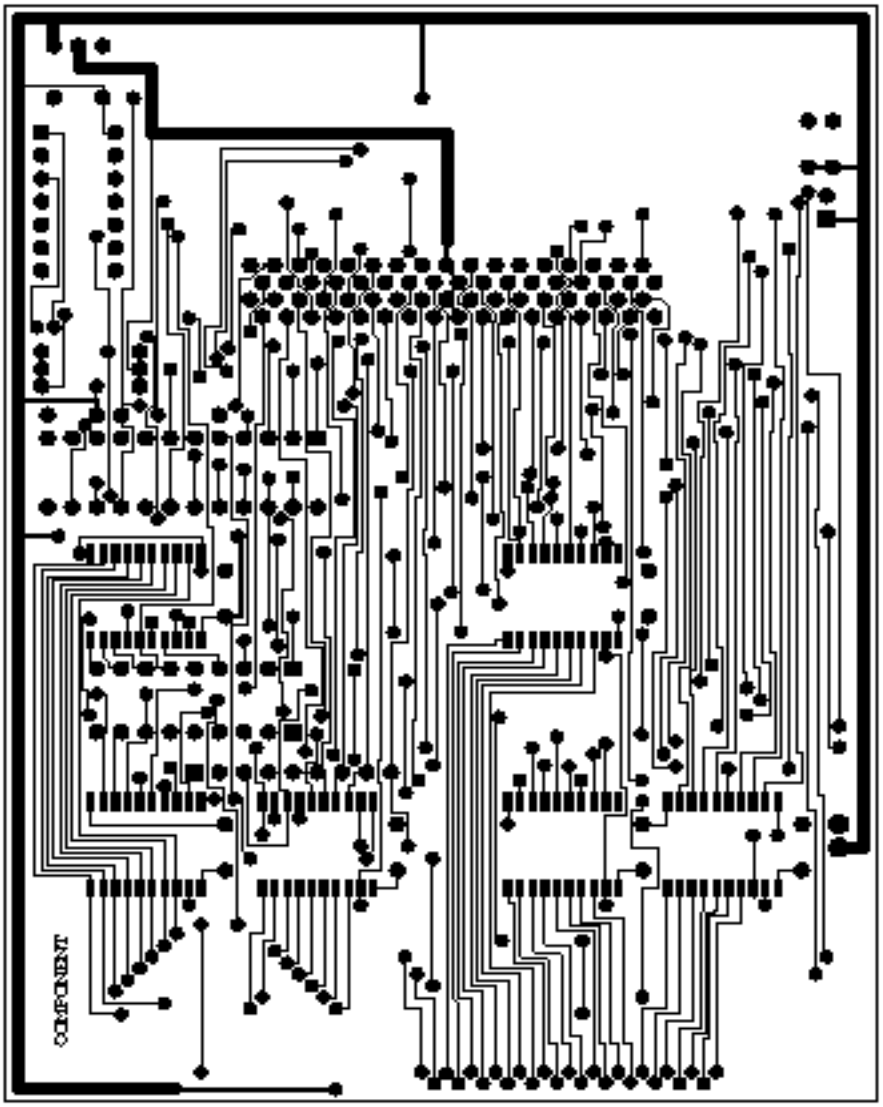
```
*X07327Y06737D01*X07127Y06737
*X07027Y06737*X05662Y06737D02
*X07027Y06737D01*X06677Y03287D02
*X07327Y03287D01*X07327Y05087
*X07327Y06587*X07202Y06587*D02
*M02*
```

Tiskano vezje vidimo na slikah 2.46 in 2.47. Vidimo, da je vezje 2-plastno, komponentna stran je na sliki 2.46. Na sliki 2.48 vidimo stop lak (zaščitno plast) tiskanega vezja za dani primer. Slika ni najboljša, saj zaradi nenatančnosti (prikazovalna grafika, ločljivost printerja) ne loči dobro sosednih delcev in okroglin, čeprav je pri normalni ločljivosti foto-izhod povsem brez omenjenih težav. Na sliki 2.49 je podan beli tisk obravnavanega tiskanega vezja.

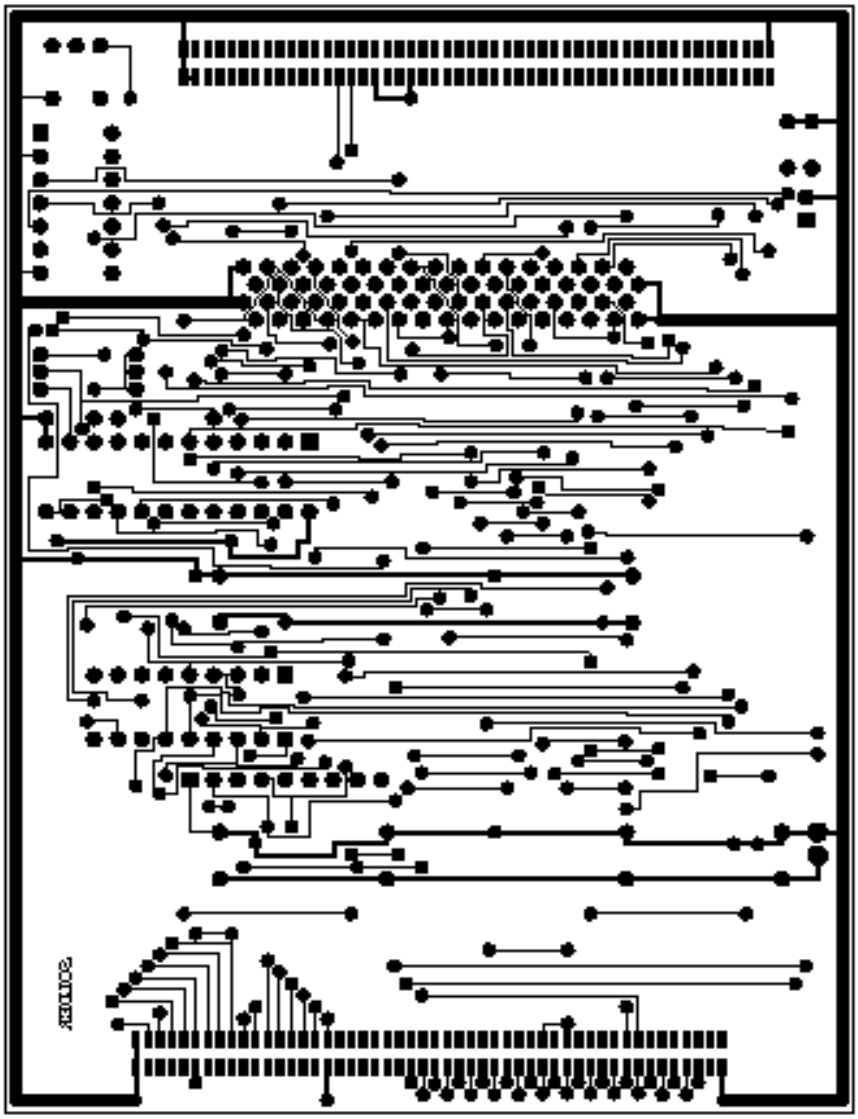
2.7.6 Tiskana vezja za posebne zahteve

Kot posebnost je potrebno omeniti tiskano vezje na keramični podlagi. Takšno vezje je sorazmerno drago, ker je posledica tehnologije izdelave. Pri "pečenju" keramike se le ta krči. Ker je sprememba v velikosti velika, je potrebna velika natančnost postopka pečenja in kvaliteta materialov. Podjetje IBM izdeluje do 50-plastna tiskana vezja na keramični osnovi.

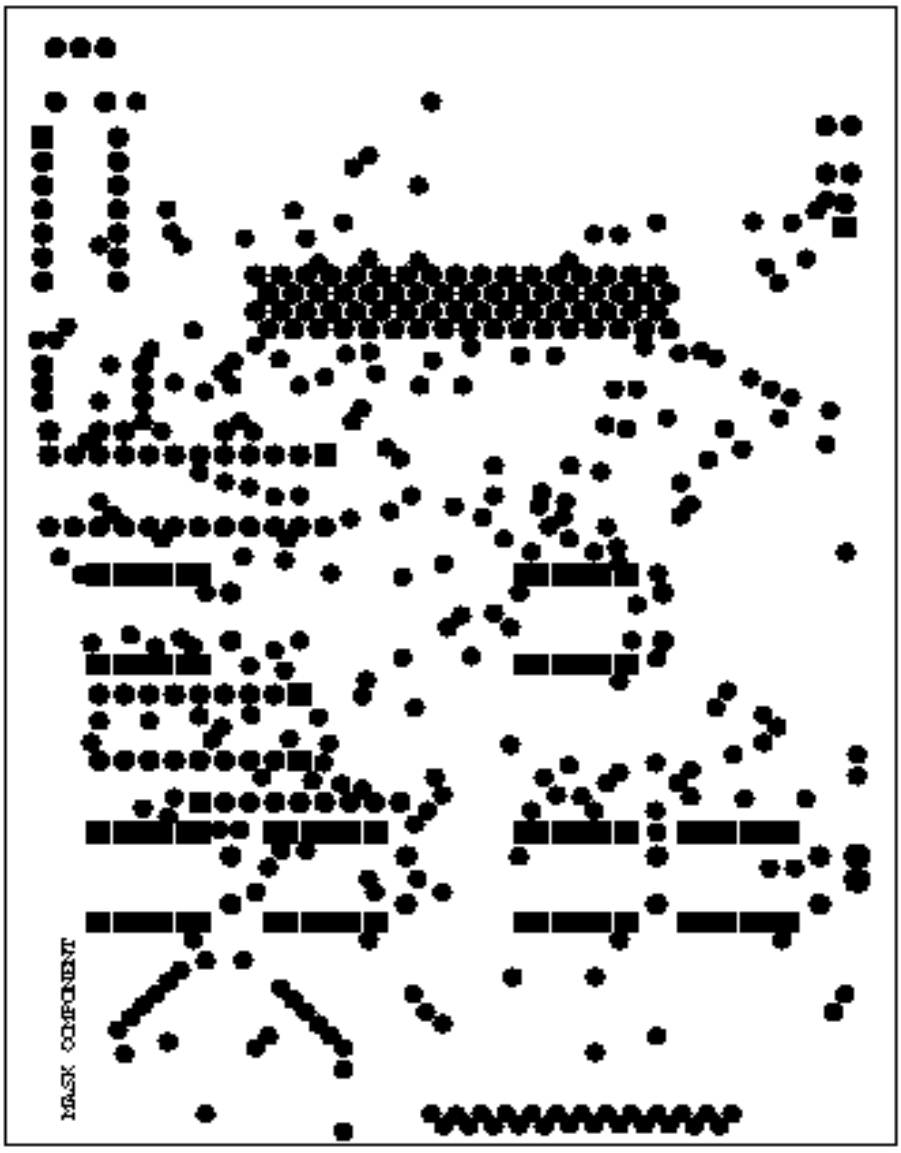
Posebnost so gibka tiskana vezja. Ta vezja pridejo prav, kjer zaradi omejitev v prostoru ne moremo uporabljati normalnih negibkih tiskanih vezij. Takšno vezje najdemo na primer v fotografskih aparatih. V tem aparatu ni veliko prostora, še manj pa ravnega prostra.



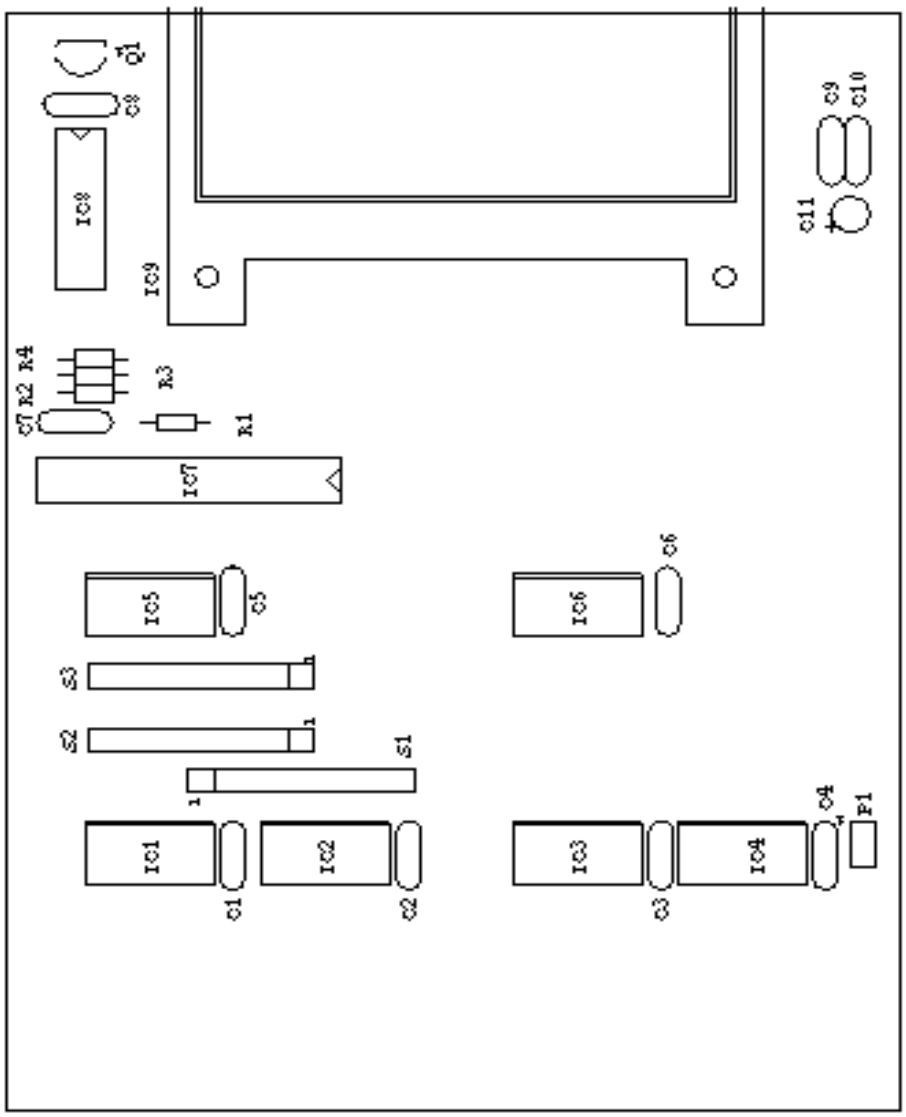
Slika 2.46: Navpični del tiskanega vezja (P-Cad).



Slika 2.47: Vodoravni del tiskanega vezja (P-Cad).



Slika 2.48: Primer za stop lak tiskanega vezja (negativ) (P-Cad).



Slika 2.49: Primer belega tiska na tiskanem vezju (P-Cad).

Poglavje 3

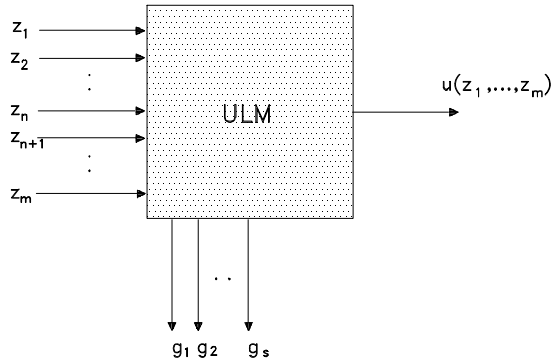
LOGIČNA MODULARNOST

3.1 UVOD

Logični modul je za nas vsako logično vezje, ki ima dovolj veliko logično vsebino, zaokroženo tako, da ima čim manjšo interakcijo s sosednimi logičnimi moduli. Moduli so lahko na osnovnem ali višjem nivoju. Strukturo logičnega modula želimo namestiti na en čip s čim manj priključki. V takšnem smislu smo se lotili modularnosti že v prejšnjem poglavju. Programirnost čipov pa pojem logičnega modula razširja, saj se kasnejšemu načrtovalcu RSS pri takšnih čipih ponuja dostopnost do notranjih logičnih modulov čipa. Načrtovalec ima tako možnost, da šele on opredeli dokončno funkcionalnost strukture čipa. Pri takšnem delu se pojavi nekaj novih pojmov, ki jih do sedaj še nismo srečali, namenjamo pa jim to poglavje. Med takšne pojme sodijo: univerzalnost logičnega modula, funkcija, ki se hkrati nanaša na logične in nelogične spremenljivke, celica, iterativne celične strukture in še kaj.

3.2 UNIVERZALNI LOGIČNI MODULI

Univerzalni logični modul ULM je modul, ki lahko da na svojem izhodu vsako logično funkcijo n neodvisnih spremenljivk. Kot vemo, je teh funkcij po številu 2^{2^n}. Univerzalni modul za $n = 2$ omogoča, da pri primerni izbiri dveh vhodov lahko ponudi vsako od 16 možnih funkcij 0, AND, OR, NAND, NOR, XOR, ..., 1. Univerzalne module lahko delimo v dve osnovni kate-



Slika 3.1: Splošna bločna shema univerzalnega logičnega modula.

goriji: navadni univerzalni in sekvenčni univerzalni moduli. Prve srečamo v decizijskih in druge v sekvenčnih (avtomatnih) RSS.

3.2.1 Navadni univerzalni moduli

Slošno shemo navadnega univerzalnega modula prikazuje slika 3.1. Vhodna spremenljivka z_i je lahko katerikoli element množice $\{0, x_i, \bar{x}_i, 1, g_1, ..., g_s\}$. g_i so pomožne funkcije (enostavne logoče povezave AND, OR, XOR, ...), ki lahko precej zmanjšajo logično vsebino ULM. x_i je neodvisna spremenljivka funkcije $f(x_1, x_2, ..., x_n)$, katero želimo dobiti v smislu univerzalne funkcije $u(z_1, z_2, ..., z_m)$ na izhodu ULM.

Iz teorije preklopnih struktur [1] izhaja, da se da vsaka logična funkcija predstaviti v popolni disjunktivni ali konjunktivni normalni obliki (PDNO, PKNO). Ni logične funkcije, ki ne bi bila izrazljiva v PDNO ali PKNO. Glede na to je PDNO in PKNO univerzalni model za vse logične funkcije nekega števila n neodvisnih spremenljivk. Če vzamemo

$$z_i = x_i\,,\ \ 1 \leqslant i \leqslant n$$

$$z_j = f_{j-n-1}\,,\ \ n+1 \leqslant j \leqslant m = n + 2^n\,,$$

vidimo, da je ULM na sliki 3.1 univerzalni modul. V primeru omenjenega ULM je $s = 0$, kar pomeni, da pomožnih funkcij g ne potrebujemo. Število potrebnih priključkov, da realiziramo funkcijo $f(x_1, x_2, ..., x_n)$ z ULM, je

$$P_0 = n + 2^n. \tag{3.1}$$

V primeru n neodvisnih spremenljivk govorimo o n-ULM. Na osnovi razčlenjevanja logičnih funkcij [1] se da dokazati, da je multipleksor ULM. Poglejmo si funkcijo treh spremeljivk v razčlenjeni obliki po spremenljivkah x_2 in x_3

$$\begin{gathered} f(x_1, x_2, x_3) = \bar{x}_2\bar{x}_3 f(x_1, 0, 0) \vee \bar{x}_2 x_3 f(x_1, 0, 1) \vee \\ x_2\bar{x}_3 f(x_1, 1, 0) \vee x_2 x_3 f(x_1, 1, 1) \end{gathered} \tag{3.2}$$

3-ULM ima 6 vhodov: $f(x_1, 0, 0)$, $f(x_1, 0, 1)$, $f(x_1, 1, 0)$, $f(x_1, 1, 1)$, x_2^w in x_3^w. w pomeni le to, da je ustrezna spremenljivka lahko v originalni ali negirani obliki.

Na osnovi n-ULM lahko pristopimo k realizaciji m-ULM, pri čemer je $m \geq n$. Poglejmo si logično funkcijo s petimi spremenljivkami, že kar v razčlenjeni obliki po spremenljivkah x_4 in x_5

$$\begin{gathered} f(x_1, x_2, x_3, x_4, x_5) = \bar{x}_4\bar{x}_5 f(x_1, x_2, x_3, 0, 0) \vee \bar{x}_4 x_5 f(x_1, x_2, x_3, 1, 0) \vee \\ x_4\bar{x}_5 f(x_1, x_2, x_3, 1, 0) \vee x_4 x_5 f(x_1, x_2, x_3, 1, 1) \end{gathered} \tag{3.3}$$

Ustrezno realizacijo z 3-ULM vidimo na sliki 3.2. Potrebujemo torej 5 primerkov 3-ULM. Zadnji 3-ULM se nanaša na razčlenitev po (3.3), medtem ko se prvi štirje nanašajo na razčlenitev po izrazu (3.2). Na osnovi slike 3.2 lahko zaključimo, da je prikazana logična shema grajena modularno, na osnovi 3-ULM. Seveda lahko sedaj vzamemo 5-ULM kot modul v modularnih shemah za logične funkcije, ki imajo $n > 5$.

Preštejmo na sliki 3.2 vhodne priključke modulov. Teh je 28, medtem ko imamo pri enem modulu na sliki 3.1, $P = 2^5 + 5 = 32 + 5 = 37$. Če bi se modul 3-ULM nahajal na enem čipu, je primer na sliki 3.2 ugodnejši. V splošnem imamo pri multipleksorskih modularnih rešitvah naslednje število priključkov

$$P = 2^{n-1} + n - 1. \tag{3.4}$$

V teoriji modularnih logičnih shem so poskušali to potrebno število priključkov še zmanjšati. Poskušali so opredeliti število a tako, da je

$$n + a < 2^{n-1} + n - 1\,. \tag{3.5}$$

Ta relacija se uveljavi, če preidemo iz n-ULM na m-QULM. QULM je univerzalni logični modul, ki uporablja pomožne funkcije $g_1, g_2, ..., g_s$ (glej sliko 3.1). Razlago za QULM si oglejmo na primeru $n = 5$ in $m = 6$. Na sliki 3.3 uporabljamo 5-ULM za realizacijo glavnine modula 6-QULM. Preostanek, ki še ostane, pokrijemo s Q-vezjem, ki ga dodajamo k omenjenemu 5-ULM. Q-vezje zavisi samo od dveh spremenljivk, zato v splošnem lahko realizira katerokoli od 16 funkcij dveh spremenljivk $g(x_5, x_6)$. Takšne funkcije g_i vodimo kot vhodne spremenljivke nazaj na vhod ULM (glej sliko 3.1). S tem na preprost način pokrijemo razliko med 6-ULM in 5-ULM. Če bi vzeli 6-ULM namesto 6-QULM, bi potrebovali več vsebine in tudi več priključkov, ki jih potrebuje realizacija funkcije $f(x_1, x_2, ..., x_6)$. Katera funkcija g pa je sploh potrebna, da gremo v QULM in ne v ULM?

Če naj velja neenačba (3.5), mora biti množica vseh mintermov $m = \{m_1,\ m_2, ..., m_t\}$ funkcije $f(x_1, x_2, ..., x_n)$, ki jo želimo modularno realizirati, pokrita z množico blokov mintermov $b = \{b_1, b_2, ..., b_a\}$, ki ima manjšo moč, $a < t$. Izhodna funkcija ULM je tedaj

$$u(z_1, z_2, ..., z_{n+a}) = \bigvee_{i=1}^{a} z_{n+i}\, b_i\,. \tag{3.6}$$

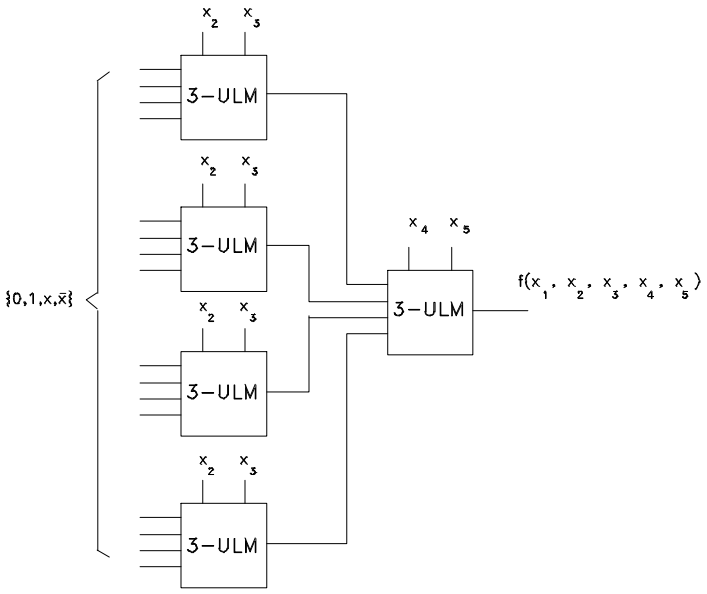
Spremenljike x_i opazovane funkcije $f(x_1, x_2, ..., x_n)$ imajo določeno izločevalno sposobnost mintermov. Za razlago vzemimo logično funkcijo

$$f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2\bar{x}_3\,. \tag{3.7}$$

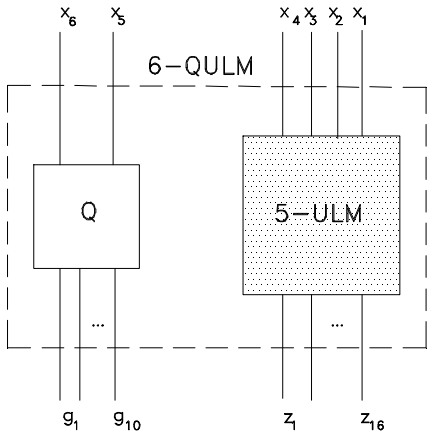
Tej pripada $m = \{m_0, m_5, m_6\}$. Posamezne spremenljivke izločajo minterme takole:

$\bar{x}_1 : \{m_0\}$
$x_1 : \{m_5, m_6\}$
$\bar{x}_2 : \{m_0, m_5\}$
$x_2 : \{m_6\}$
$\bar{x}_3 : \{m_0, m_6\}$
$x_3 : \{m_5\}$.

Če se v takšnem spisku nahajajo vse možne podmnožice množice m, je ustrezen modul v smislu izraza (3.6) ULM. V primeru da ne pridemo do vseh



Slika 3.2: 5-ULM realiziran s 5×3-ULM.



Slika 3.3: Logični modul 6-QULM.

podmnožic, spremenljivke x_i nimajo zadostne moči pri izločanju mintermov. Tedaj si pomagamo s pomožnimi funkcijami g_i. Katera funkcija g_i je pravzaprav potrebna, zavisi od tega, do katerih podmnožic mintermov moramo še priti. Funkcijo (3.7) zamenjajmo s funkcijo

$$f(x_1, x_2, x_3) = \bar{x}_1\bar{x}_2\bar{x}_3 \vee \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_2x_3 \vee x_1x_2\bar{x}_3 \tag{3.8}$$

Nove izločitve mintermov so naslednje:

$\bar{x}_1 : \{m_0, m_1\}$
$x_1 : \{m_5, m_6\}$
$\bar{x}_2 : \{m_0, m_1, m_5\}$
$x_2 : \{m_6\}$
$\bar{x}_3 : \{m_0, m_6\}$
$x_3 : \{m_1, m_5\}$.

Vse podmnožice množice m pa so:

$\{m_0\}, \{m_1\}, \{m_5\}, \{m_6\}$
$\{m_0, m_1\}, \{m_0, m_5\}, \{m_0, m_6\}, \{m_1, m_5\}, \{m_1, m_6\}, \{m_5, m_6\}$
$\{m_0, m_1, m_5\}, \{m_0, m_5, m_6\}, \{m_1, m_5, m_6\}$.

Vidimo, da spremenljivke ne izločijo podmnožice $\{m_0\}$. Če bi rabili to podmnožico, lahko pridemo do nje na primer s funkcijo $g = \bar{x}_1\,\bar{x}_3$, saj imamo v njenem primeru izločanje

$\{m_0, m_1\} \cap \{m_0, m_6\} = \{m_0\}$.

Funkcijo q peljemo nazaj na vhod ULM, s čemer ji omogočamo, da izloča minterme.

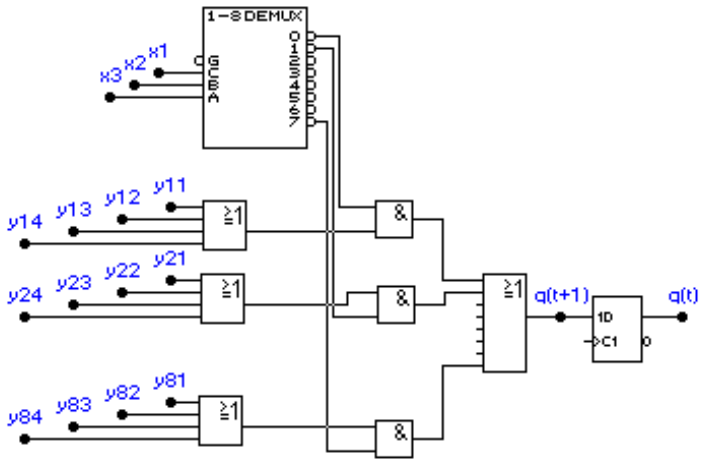
Če sliko 3.3 posplošimo, vidimo, da če vstopa v Q-vezje r spremenljivk, vstopa v TULM $n - r$ spremenljivk. Na izhodu Q-vezja je tako lahko $2^{2^r} - (2r + 2)$ in na izhodu TULM 2^{n-r} funkcij. Vseh priključkov v primeru QULM je tako

$$P = 2^{2^r} + 2^{n-r} - 2r - 2. \tag{3.9}$$

Normalno je r majhno število, do $n = 10$ jemljemo $r = 2$.

3.2.2 Sekvenčni univerzalni moduli

Iz teorije sekvenčnih preklopnih vezij izhajajo pomnilne celice D, RS, T, JK .... Vsaka od teh celic je univerzalna, kar pomeni, da ni sekvenčnega dogodka, ki ga ne bi mogli obvladati s katerimkoli tipom omenjenih pomnilnih celic.



Slika 3.4: Sekvenčni ULM z vhodno razvejanostjo in razvejanostjo v notranjost logične sheme.

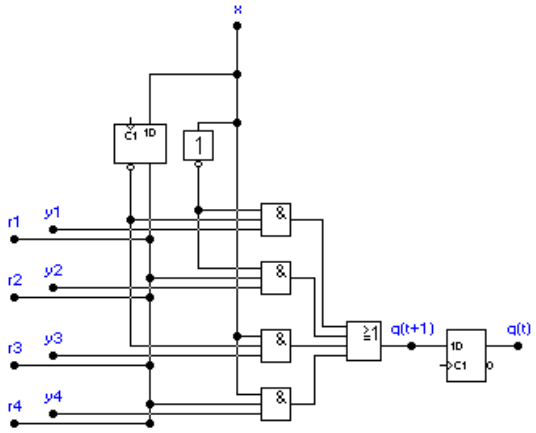
Na sliki 3.4 vidimo sekvenčni ULM za 3 vhodne spremenljivke $x_i, 4 \times 8$ notranje spremenljivke y_{ij} in enim izhodom $q(t+1) = \mathrm{D}^1 q$. Pomnilna celica D na izhodu modula je tisti del logičnega vezja, ki vnaša v modul pomnjenje. V splošnem imamo p vhodnih spremenljivk in r notranjih spremenljivk. Na osnovi slike 3.4 sklepamo, da v splošnem primeru velja naslednja izhodna funkcija

$$\begin{aligned} \mathrm{D}^1 q &= \bigvee_{i=1}^{2^p} \left(\bigvee_{j=1}^{r} y_{ji} \right) m_i \\ m_i &= \&(x_1, x_2, ..., x_p)\,. \end{aligned} \tag{3.10}$$

Demultipleksor ustvarja naslovne argumente (minterme) m_i. Pri določeni vhodni črki so naslovljena samo ena vrata AND, ki vplivajo na izhodno D-celico. Število potrebnih priključkov za splošni modul, to je (3.10), je

$$P = r\, 2^p + p\,. \tag{3.11}$$

Če je $p = r = 1$ in $y = q$ (povratek q-ja nazaj na opazovani modul), nam (3.10) da najenostavnejši izhod



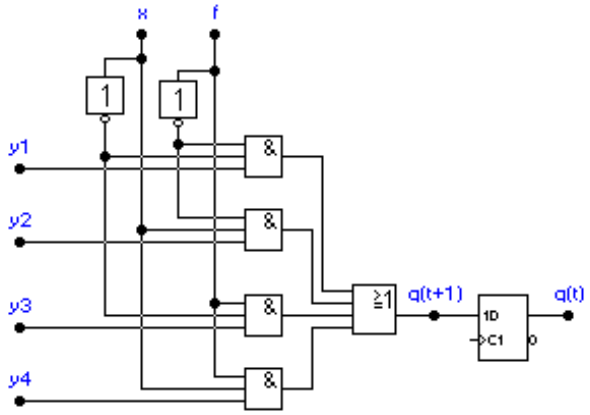
Slika 3.5: Drevesni sekvenčni ULM.

$$D^1 q = y_{11}\bar{x} \vee y_{13}x = q\bar{x} \vee \bar{q}x,$$

kar je izhodna funkcija navadne T-pomnilne celice.

Naslednji sekvenčni ULM je povratni ULM. Vidimo ga na sliki 3.5. Poleg izhodne D-celice imamo D-celico tudi na vhodni strani modula. Spremenljivki x in $\mathrm{D}^{-1}x$ nam dasta časovno pogojene 4 naslove za odpiranje 4 vrat AND. Posebnost tega modula je, da iz notranjega okolja dobiva notranje spremenljivke y, v notranje okolje pa tudi pošilja spremenljivke r. Izhodna funkcija takšnega modula je

$$\begin{aligned}
&\mathrm{D}^t q = \bigvee_{i=0}^{2g} h_i\, m_{ti} \\
&m_{ti} = \&(\mathrm{D}^{t-1}x, \mathrm{D}^{t-2}x, ..., \mathrm{D}^{t-g})\ , \quad t \geq g \\
&m_{ti} = \&(\mathrm{D}^{t-1}x, \mathrm{D}^{t-2}x, ..., \mathrm{D}^{0})\ , \quad t < g \\
&\mathrm{D}^t q = 0 \text{ ali } \mathrm{D}^t q = 1\ , \quad t = 0\,.
\end{aligned} \tag{3.12}$$



Slika 3.6: Sekvenčni ULM s povratno funkcijo f.

Modul je dobil atribut drevesni zato, ker na njegovi osnovi lahko načrtujem drevesne RSS.

Naslednji sekvenčni modul je izpeljanka iz modula na sliki 3.5. Razlika je v tem, da v modulu ne nastopa več D-celica na vhodni strani modula. Tega nadomestimo s povratno funkcijo f, ki jo realiziramo posebej, izven modula, tako, da imamo opraviti z dvem vhodnima spremenljivkama x in f. Zaradi povratne funkcije modul imenujemo povratni sekvečni ULM. Vhode y lahko tudi v primeru tega modula disjunktivno razpremo (disjunkcija z r vhodi). Tudi spremenljivko x lahko demultipleksiramo. Splošna izhodna funkcija takšnega modula je

$$\begin{aligned} \mathrm{D}^1 q &= \bigvee_{i=1}^{2^p} \left(\bigvee_{i=1}^{2^p} y_{ji} \right) m_i \\ m_i &= \&(x_1, x_2, ..., x_p, f). \end{aligned} \tag{3.13}$$

3.3 MODULARNA SHEMA AVTOMATA

Kako načrtujemo decizijske modularne sheme, smo že spoznali v okviru slike 3.2. V tem razdelku si oglejmo, kako pristopiti k avtomatu na modularni

način. Seveda moramo imeti na voljo sekvenčne univerzalne module sicer modularno avtomata ne moremo realizirati.

Da lahko načrtujemo modularne sheme z obdelanimi ULM, moramo predhodno formalno definirati modularno shemo. Modularna shema je petorka

$$V_{rp} = \{M_j, X, m_i, f, M\}. \tag{3.14}$$

Pri tem je $M_j = \{m_1, m_2, ..., m_k\}$ množica imen modulov, ki so v modularni shemi V_{rp}. $X = \{x_1, x_2, ..., x_{2^p}\}$ je vhodna abeceda in m_i je ime izhodnega modula. V množici M so tisti moduli, ki imajo v začetnem stanju vrednost 1. f je funkcija porojevanja predhodnih modulov. Za njo velja

$$\begin{aligned} &f : X \times M_j \rightarrow \{m_1, m_2, ..., m_w\} \cup M^* \\ &w \leq r,\ m_j \in M_j,\ M^* = \{0, 1\}. \end{aligned} \tag{3.15}$$

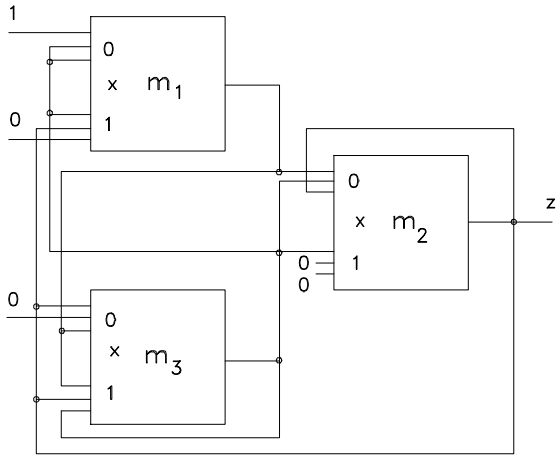
Spremenljivka r pove, kakšno je število notranjih spremenljivk na eno vhodno kombinacijo (razvejanost modula za notranje spremenljivke), medtem ko je p število primarnih vhodnih spremenljivk. Število možnih vhodnih kombinacij je 2^p. Velikost modularne sheme V_{rp} presojamo s številom (V_{rp}), ki je enako številu vseh modulov v shemi.

Zgled 3 *:*

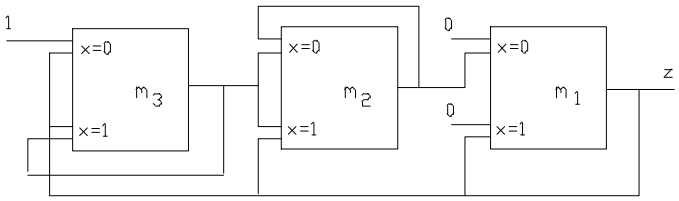
Za sekvenčno modularno shemo na sliki 3.7 *izpišimo funkcijo generiranja predhodnih modulov.*

$$\begin{aligned} &f(0, m_1) = 1 \in M^*,\ 1 \vee y_2 \vee y_3 = 1 \\ &f(1, m_1) = \{m_2, m_3\} \\ &f(0, m_2) = \{m_1, m_2, m_3\} \\ &f(1, m_2) = \{m_3\} \\ &f(0, m_3) = \{m_1, m_2\} \\ &f(1, m_3) = \{m_1, m_2, m_3\}. \end{aligned}$$

Od vseh možnih podmnožic funkcija f ne generira podmnožice modulov $\{m_1\}$, $\{m_2\}$ *in* $\{m_1, m_3\}$.



Slika 3.7: Modularna shema V_{31}.



Slika 3.8: Sekvenčna modularna shema V_{21}.

Za sekvenčne modularne sheme so predvsem pomembni dogodki (regularni izrazi [1]), ki se odvijajo na njihovih vhodih, notranjosti in izhodih. Kako odčitavamo delovne sekvence sheme, si oglejmo na primeru slike 3.8. Imamo le eno vhodno spremenljivko in razvejanost modula $r = 2$. Izhodni modul je m_1, popolna definicija sheme pa je

$$V_{21} = \{M_j = \{m_1, m_2, m_3\}, X = \{0, 1\}, m_i = m_1, f, M = \{m_3\}\}.$$

Iz te definicije izhaja, da je v začetku aktiven samo modul m_3. Začetno stanje je tako $s_0 = 001$. Iz tega začetnega stanja gremo v tabeli 3.1 v stanje s_1, s_2, s_3 tako, da stanje s_i ustreza času $t + i$. Prehajanje stanj seveda zavisi

Stanje	t	x	m_1	m_2	m_3
s_0	$t+1$	-	0	0	1
s_1	$t+2$	0	0	1	1
s_2	$t+3$	1	0	1	1
s_3	$t+4$	0	1	1	1

Tabela 3.1: Določanje izhodne sekvence sekvenčne modularne sheme.

od notranjih spremenljivk in primarne vhodne spremenljivke x z vhodnim dogodkom $\aleph = -010$. V tem primeru simbol $-$ pomeni, da ni pomembno, katera vhodna črka je pri začetnem stanju modularne sheme.

Tabela 3.1 omogoča splošnejši (obči) zapis procesiranja vhodnega dogodka, kot je to običaj v avtomatni teoriji. Naj je $x_1 = (x = 0)$ in $x_2 = (x = 1)$ ter $a_1 = (m_1m_2m_3 = 001)$, $a_2 = (m_1m_2m_3 = 011)$, $a_3 = (m_1m_2m_3 = 111)$. Pri vhodni sekvenci $\aleph = x_1x_2x_1$ imamo na izhodu sekvenco $\Im = a_2a_2a_3$. Začetno stanje ne uvrščamo v delovne sekvence, ker je lahko prvo stanje, ki lahko zazna vhodne razmere, šele stanje po začetnem stanju (Mooreov avtomat).

Z računanjem velikosti sekvenčnih modularnih shem pri danih dogodkih nad vhodno abecedo X se je ukvarjal J. D. Ulman [10]. Ta je določil velikost sheme za determiniran in nedeterminiran avtomat. Determiniran avtomat je tak avtomat, ki ima povsem definirano naslednje stanje pri vsakem prehajanju. Naslednje stanje je vedno eno in edino. Nedeterminiran avtomat ima za vsaj eno naslednje stanje (na vsaj enem preseku stolpca in vrstice tabele prehajanja stanj) podmnožico stanj, ki vsebuje več kot eno stanje. Zadnje pomeni, da vemo, v kateri podmnožici je avtomat nahaja, ne vemo pa, v katerem elementu te podmnožice. V principu lahko tudi rečemo, da se nedeterminiran avtomat nahaja hkrati v več stanjih.

V shemah na slikah 3.7 in 3.8 lahko opazimo, da imamo nekaj vhodov, na katere vežemo konstanti 0 in 1. Možno je zasnovati takšne modularne sekvenčne sheme, ki so enic proste (1-free network). Za njih velja, da nobena preslikava $f(x, m)$ ne more dati vrednost iz množice M^*.

Velikost sheme za realizacijo determiniranega avtomata je

$$(V_{rp}) = \left(\frac{2^p r}{2^p r - 1} + \frac{2^p}{2^p - 1}\right) n^{1+p\log_r 2}\ ,\quad r \geq 2 \tag{3.16}$$

in za realizacijo nedeterminiranega avtomata

$$(V_{rp}) = \frac{2^p r}{2^p r - 1} + (n+1)\, n^{1+p\log_r 2}\ ,\ \ r \geq 2\,. \tag{3.17}$$

Vezja V_{rp} so pri tem enic-prosta vezja, p je število primarnih vhodnih spremenljivk, r disjunktivna razvejanost modulov, t. j. sekundarnih vhodov y in n število stanj avtomata. V primeru izrazov (3.16) in (3.17) govorimo o r-omejenih avtomatih.

Naj je A nedeterminiran avtomat s k stanji. Pri teh stanjih in vhodni abecedi X so možni določeni dogodki. Modularna shema V lahko ekvivalentno pokrije te dogodke, če ima velikost $(V) = k$. Velikost sheme V, ki realizira avtomat A, lahko izračunamo tudi drugače. Vzemimo, da je $< M_j >$ množica vseh možnih podmnožic množice modulov M_j. Vrednosti funkcij $f(x, m)$ spadajo v množico $< Mj >$. Ker je v tej množici največ 2^k elementov, ne more biti v modularni shemi več kot 2^k modulov. V primeru da je element množice $< Mj >$ sama množica M_j (v avtomatu A imamo tedaj opraviti s kompletno množico stanj, notranjo abecedo), v shemi V ni potreben noben modul, saj je pri opisovanju dogodkov primerno vsako stanje iz notranje abecede avtomata A. Če odštejemo to možnost in prazno množico modulov, dobimo oceno

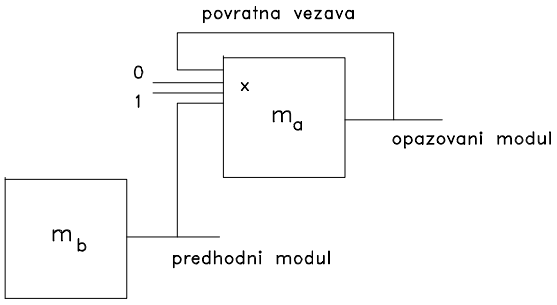
$$(Vrp) \leq 2^k - 2\,. \tag{3.18}$$

Oglejmo si sedaj postopek, kako bi realizirali nemodularno podan avtomat A z modularno shemo V, če v V nastopajo moduli po sliki 3.4. Naj ima avtomat A izhodno črko $z \in \{0, 1\}$ (bolj kompliciranе izhodne črke so dvojiške kode). V tem primeru lahko zapišemo particijo P_a

$$P_a = \{b_0, b_1\}, \tag{3.19}$$

v kateri je za blok b_0 značilna izhodna črka $z = 0$ in za blok b_1 črka $z = 1$. Prvi modul, ki ga izberemo v shemi V_{rp}, je izhodni modul m_i. Temu ustreza podmnožica stanj avtomata A, ki so v bloku b_1. Ko je m_i že uveden, lahko pristopimo k predhodnemu modulu na osnovi generiranja modulov

$$f(x, m_i) = m_j,\ x \in X,\ m_i \equiv b_1 \tag{3.20}$$



Slika 3.9: Povezovanja in zaključevanja modulov v sekvenčni modularni shemi.

Če dobimo $m_j = m_i$, novega modula ne potrebujemo, saj je prišlo do povratne vezave, izhod modula m_i je vezan nazaj na vhod istega modula. Naj modulu m_j ustreza v avtomatu A blok b_j, $m_j \equiv b_j$. V kolikor je ta blok prazna množica, damo na opazovani vhod modula m_i Booleovo konstanto 0, in v kolikor je ta blok polna notranja abeceda (vsa stanja), damo na opazovani vhod Booleovo konstanto 1. V vseh drugih primerih damo na vhod modula z b_j to, kar nam ponudi $f(x, m_j)$ itd., vse gledano v globino modularne sheme. Blok b_j lahko odčitamo v tabeli prehajanja stanj avtomata A, če vzamemo prihajajočo prihodnost $(t+1)$ za sedanjost (t) itd. Vse štiri omenjene možnosti pri prehodu iz modula na modul v smeri proti začetku modularne sheme vidimo na sliki 3.9. Ko pridemo samo do povratnih vezav, konstant 0 in 1, smo na začetku modularne sheme V. Zadnje jemljemo kot konec načrtovanja sheme.

Zgled 4 *:*

Dan je avtomat s prehajanjem stanj po tabeli 3.2. *Načrtajmo ustrezno modularno shemo* V_{12}.

Avtomat ima tri stanja s*, štiri vhodne črke* x *in dve izhodni črki* z*. Pri štirih vhodnih črkah je* $p = 2$*. Podana notranja razvejanost* $r = 1$ zahteva za vsako vhodno črko $x_i, i = 1, 2, 3, 4$ *samo en kanal za spremenljivke* y*. Izhodni modul oziroma ustrezeni blok particije nam mora dati vrednost 1 samo v primeru, ko je* $z_2 = 1$*. Tako je podmnožica stanj s to značilnostjo* $b_1 = \{s_2, s_3\}$*, kar bomo krajše pisali kar* $m_1 \equiv b_1 = \{2, 3\}$*. Na osnovi tabele* 3.2 *postavimo drevo tako, da nam ta pove, kakšna je preteklost (pomnilna*

z	z_1	z_2	z_2
$x \setminus s$	s_1	s_2	s_3
x_1	s_2	s_2	s_3
x_2	s_1	s_1	s_3
x_3	s_2	s_3	s_1
x_4	s_1	s_3	s_2

Tabela 3.2: Primer Mooreovega avtomata.

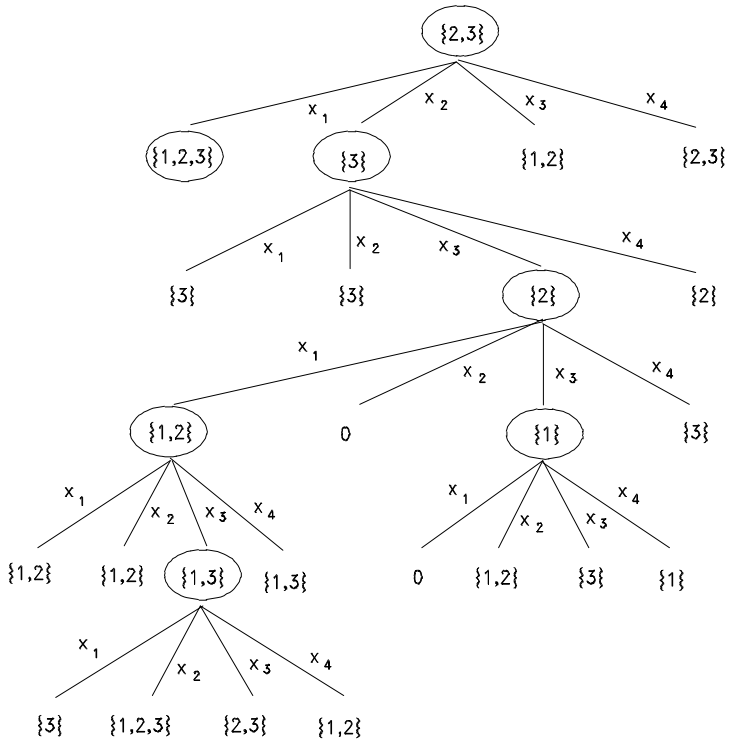
globina) v avtomatu, pod pogojem, da imamo v sedanjosti opraviti z blokom b_1 *.To drevo prikazuje slika* 3.10*. Vrh drevesa predstavlja blok* $b_1 = \{2, 3\}$*, vsi ostali bloki izhajajo iz pohoda v preteklost v tabeli* 3.2*. Pri vhodni črki* x_1 *se pri stanju* s_2 *ali* s_3 *nahajamo predhodno v stanju* s_1 *ali* s_2 *ali* s_3*, zato je predhodna množica* $\{1, 2, 3\}$*. Pri vhodni črki* x_2 *se pri stanju* s_2 *ali* s_3 *nahajamo lahko le v stanju* s_3*, zato je pred nami predhodni blok oziroma množica* $\{3\}$*. Itd. Kje se to drevo končuje? S postopkom ne moremo več nadaljevati, ko pridemo do popolne notranje abecede* $\{1, 2, 3\}$*, prazne množice* 0 *ali do podmnožice, ki smo jo že obravnavali. Prazna množica je v drevesu že označena z* 0*, polno množico (prostor vseh stanj) pa označimo z* 1*. Obe konstanti sta elementa množice* $M^* = \{0, 1\}$*.*

Obkroženi bloki stanj na sliki 3.10 *nam definirajo vse možne, po funkciji različne module v* V_{12}*, kot sledi:*

$$
\begin{aligned}
&f(x_1, \{2, 3\}) = 1 \in M^* \\
&f(x_2, \{2, 3\}) = \{3\} \\
&f(x_3, \{3\}) = \{2\} \\
&f(x_1, \{2\}) = \{1, 2\} \\
&f(x_3, \{1, 2\}) = \{1, 3\} \\
&f(x_3, \{2\}) = \{1\} \\
&f(x_2, \{2\}) = 0 \in M^*.
\end{aligned}
$$

Vsaki dobljeni podmnožici avtomata dodelimo ime ustreznega modula

$$
\begin{aligned}
&\{2, 3\} \equiv m_1 \\
&\{3\} \equiv m_2 \\
&\{1, 2\} \equiv m_3 \\
&\{1\} \equiv m_4 \\
&\{2\} \equiv m_5 \\
&\{1, 3\} \equiv m_6 \,.
\end{aligned}
$$



Slika 3.10: Drevesna struktura dogajanj v avtomatu s tabelo prehajanja stanj 3.2.

Iz podanega drevesa izhajajo naslednje funkcije generiranja modulov (povezav med moduli) modulske sheme V_{12}*:*

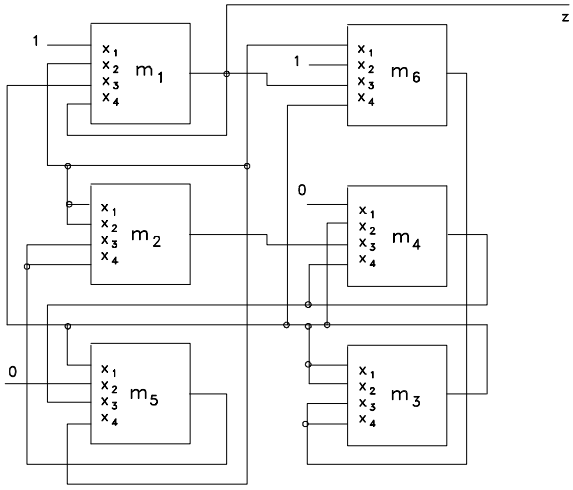
$$
\begin{aligned}
&f(x_1, m_1) = 1 \in M^* \\
&f(x_2, m_1) = m_2 \\
&f(x_3, m_1) = m_3 \\
&f(x_4, m_1) = m_1 \\
&f(x_1, m_2) = m_2 \\
&f(x_2, m_2) = m_2 \\
&f(x_3, m_2) = m_5 \\
&f(x_4, m_2) = m_5 \\
&f(x_1, m_3) = m_3 \\
&f(x_2, m_3) = m_3 \\
&f(x_3, m_3) = m_6 \\
&f(x_4, m_3) = m_6 \\
&f(x_1, m_4) = 0 \in M^* \\
&f(x_2, m_4) = m_3 \\
&f(x_3, m_4) = m_2 \\
&f(x_4, m_4) = m_4 \\
&f(x_1, m_5) = m_3 \\
&f(x_2, m_5) = 0 \in M^* \\
&f(x_3, m_5) = m_4 \\
&f(x_4, m_5) = m_2 \\
&f(x_1, m_6) = m_2 \\
&f(x_2, m_6) = 1 \in M^* \\
&f(x_3, m_6) = m_1 \\
&f(x_4, m_6) = m_3 \,.
\end{aligned}
\tag{3.21}
$$

Na osnovi funkcij (3.21) *lahko povežemo module v modularno shemo, ki jo vidimo na sliki* 3.11.

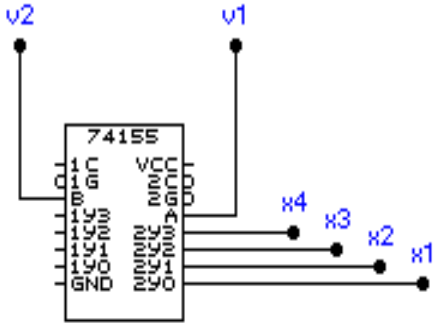
K*ako lahko pridemo do štirih vhodnih črk pri* $p = 2$*, vidimo na sliki* 3.12*. Izbran je en od dveh demultipleksorjev na čipu* 74155*. Za primer* H*-logike je potrebno izhode* $x_1, ..., x_4$ *še negirati.*

3.4 REALIZACIJA RSS S POVRATNIMI ULM

Na sliki 3.5 smo spoznali povratni ULM, ki omogoča gradnjo sekvenčnih struktur v obliki drevesa. Modul vsebuje dve pomnilni celici: ena celica je



Slika 3.11: Modularna shema danega Mooreovega avtomata.



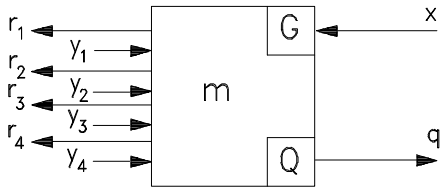
Slika 3.12: Prehod iz vhodnih spremenljivk v_1 in v_2 na vhodne črke $x_1, ..., x_4$.

na vhodni strani za postavljanje vhodnih imen za štiri vrata vrata AND, druga celica je na izhodu logične sheme modula. Naj ima prva začetno stanje G in druga, začetno stanje Q. Izhodna funkcija modula, ki upošteva te pogoje, je

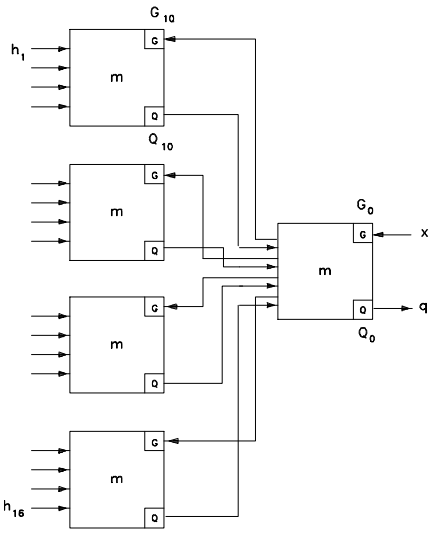
$$\begin{aligned} &\mathrm{D}^t q = \mathrm{D}^{t-1}(y_1\bar{x})\,\mathrm{D}^{t-2}\bar{x} \vee \mathrm{D}^{t-1}(y_2\bar{x})\,\mathrm{D}^{t-2}x \vee \\ &\vee \mathrm{D}^{t-1}(y_3 x)\mathrm{D}^{t-2}\bar{x} \vee \mathrm{D}^{t-1}(y_4 x)\,\mathrm{D}^{t-2}x \ , \ t \geq 2 \\ &\mathrm{D}^1 q = y_1\bar{x}\bar{G} \vee y_2\bar{x}G \vee y_3 x\bar{G} \vee y_4 xG \ , \ t = 1 \\ &\mathrm{D}^0 q = q = Q \ , \ t = 0\,. \end{aligned} \tag{3.22}$$

Pri sekvenčnih RSS je vedno tudi pomembno, kakšna je njena pomnilna globina g [11]. Če je $g = 0$, nimamo nobenega pomnjenja, zato v ustreznih logičnih shemah shajamo brez pomnilnih celic. V primeru globine $g = 1$, nam zadostuje izhodna funkcija ene pomnilne celice, v kateri, kot vemo, nastopajo spremenljivke $\mathrm{D}^0 a = a$, $\mathrm{D}^{-1}a$ oziroma $\mathrm{D}^{+1}a$ (a je poljubna spremenljivka), zavisno od tega, ali opazujem celico na vhodni ali izhodni strani. V tem primeru nam zadošča en ULM, za katerega vidimo bločno shemo na sliki 3.13. V primeru večje pomnilne globine, $g > 1$, takšne module sestavljamo v sekvenčna vezja, pri katerih velja izhodna funkcija (3.22).

Za pomnilno globino $g = 2$ je na mestu modularna shema na sliki 3.14. $g = 2$ med drugim pomeni tudi to, da ima modularna shema dva nivoja. Izhodna funkcija splošne drevesne modularne sheme na osnovi modulov na sliki 3.13 je



Slika 3.13: Bločna shema drevesnega modula.



Slika 3.14: Drevesna modularna shema s pomnilno globino $g = 2$.

$$
\begin{aligned}
&\mathrm{D}^t q = \bigvee_{i=1}^{2^{2g}} h_i \, m_{ti} \ , \ t \geqslant 2g \\
&m_{ti} = \&(\mathrm{D}^{t-1}x, \mathrm{D}^{t-2}x, ..., \mathrm{D}^{t-2g}x) \ , \ t \geqslant 2g \\
&\mathrm{D}^t q = \bigvee_{i=1}^{2^{2t}} h_i \, m_{ti} \ , \ g \leqslant t < 2g \\
&m_{ti} = \&(\mathrm{D}^{t-1}x, \mathrm{D}^{t-2}x, ..., x, 0, 0, ..., 0) \ , \ g \leqslant t < 2g \\
&\mathrm{D}^t q = \bigvee_{i=1}^{2^{2t}} Q_{ti} \, m_{ti} \ , \ 0 < t < g \\
&m_{ti} = \&(\mathrm{D}^{t-1}x, \mathrm{D}^{t-2}x, ..., x, 0, 0, ..., 0) \ , \ 0 < t < g \, .
\end{aligned}
\tag{3.23}
$$

Zaključek, da je podana izhodna funkcija primerno postavljena, lahko napravimo pri $g = 2$ in shemi na sliki 3.14. Že po drugem nivoju pridemo do vseh 16 neodvisnih vhodnih kostant h_i. Na prvem nivoju sta začetna pogoja zapisana z enomestnim in na drugem z dvomestnim indeksom. Izhodna

funkcija za shemo na sliki 3.14 je

$$\begin{aligned} \mathrm{D}^0 q &= Q_0 \\ \mathrm{D}^1 q &= Q_{10}\, m_{ti}(x, G_0) \\ \mathrm{D}^2 q &= \bigvee_{i=1}^{16} h_i\, m_{ti}(\mathrm{D}^1 x, x, G_0, G_{10})\,. \end{aligned} \tag{3.24}$$

Velikost modularne sheme pri dani pomnilni globini g je po Newbornu in Arnoldu [11]

$$(V) = \frac{(4^g - 1)}{3}\,. \tag{3.25}$$

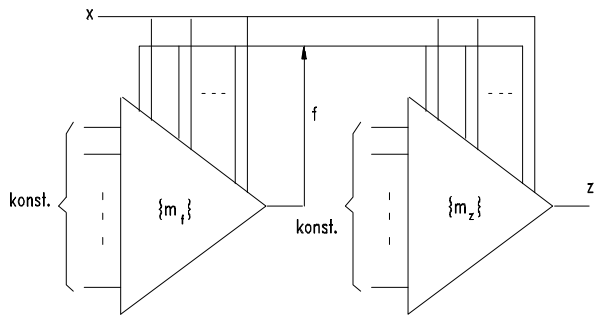
3.5 POSTAVITEV POVRATNE FUNKCIJE V AVTOMATU

Oglejmo si še načrtovanje sekvenčne modularne sheme z moduli na sliki 3.6, ki zahtevajo povratno funkcijo f. V tem razdelku poskusimo odgovoriti predvsem na vprašanje, zakaj je ta funkcija sploh potrebna v modularnih shemah. Modularne sheme, ki uporabljajo funkcijo f, so relativno obsežne. To trditev potrjuje bločna shema na sliki 3.15, ki zahteva dve mudularni shemi za opazovani avtomat, čeprav bi pri tipu modula na sliki 3.5 za ta isti avtomat zadostovala le ena modularna shema (na sliki 3.15 desna, z izhodno črko z). Poleg modularne strukture za izhodno črko z je torej potrebna še modularna shema za funkcijo f. Vidimo, da sta primarni vhodni spremenljivki x in f potrebni v obeh drevesnih shemah.

Postopek za postavitev obeh drevesnih shem je enak, kar ugodno vpliva na čas načrtovanja omenjenih sekvenčnih modularnih shem V. Izhodna funkcija povratne modularne sheme je relativno preprosta

$$\mathrm{D}^1 q = y_1 \bar{x} \bar{f} \vee y_2 \bar{x} f \vee y_3 x \bar{f} \vee y_4 x f\,. \tag{3.26}$$

Kako računamo z (3.26), si oglejmo v okviru avtomata, ki je podan s tabelo prehajanja stanj 3.2. Slika 3.10 prikazuje drevo dogajanj v omenjenem avtomatu. To drevo lahko podamo v obliki diagrama prehajanja stanj, če za



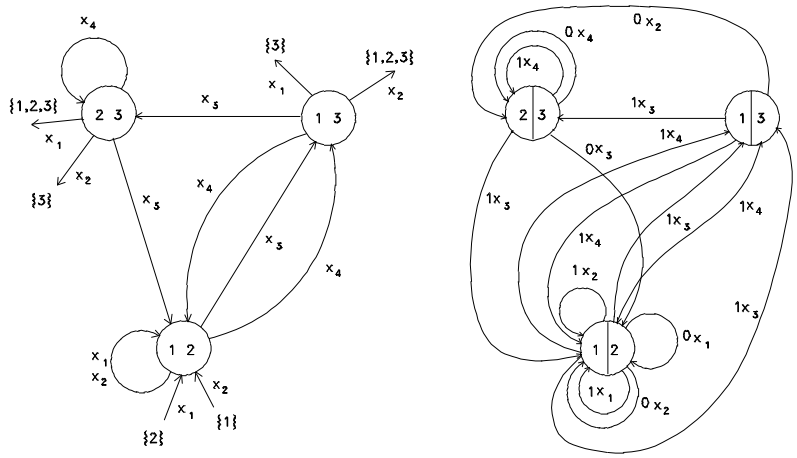
Slika 3.15: Skelet sekvenčne modularne sheme s povratno funkcijo f.

stanja modularne sheme vzamemo podmnožice stanj avtomata, ki jih predstavljajo moduli v modularni shemi. Tako na primer modul m_2 oziroma ustrezno modulsko stanje $\{3\}$ predstavlja samo avtomatovo stanje s_3 in zato ni problematičen pri postavljanju diagrama prehajanja stanj. Drugače je na primer pri modulu $m_3 \equiv \{1, 2\}$, ki predstavlja dve avtomatovi stanji, stanji s_1 in s_2. Če vstopamo v množico stanj $\{s_1, s_2\}$ z vidika nemodularnosti, ne vemo, ali vstopamo v stanje s_1 ali v stanje s_2. Modularna shema na sliki 3.11 je povsem opredeljena pri izbranem tipu modulov, takšno opredeljenost pa v primeru modularne sheme po modelu na sliki 3.15 dosežemo šele s povratno funkcijo f. Ta poveča število vhodnih črk avtomata, seveda zato, da odpravi vse neopredeljenosti pri prehajanjih stanj avtomata.

Vsa kritična stanja v primeru drevesa na sliki 3.10 so $\{2, 3\}$, $\{1, 3\}$ in $\{1, 2\}$. Stanje $\{1, 2, 3\}$ ni kritično, saj v tem primeru ni nobene selektivnosti med stanji (gre za množico vseh možnih stanj avtomata, zato v modularnih shemah uporabljamo enoto, Booleovo konstanto 1). Pri kritičnih stanjih so pravzaprav problematične samo povratne zanke pri avtomatnih stanjih. Če s funkcijo f prekinemo vse zanke, je pred nami rešitev za modularno shemo s povratno funkcijo f.

V našem primeru imamo avtomat z dvema izhodnima črkama. Če bi jih imeli več, bi bilo potrebno vzeti tudi več funkcij f. Več kot 2 izhodni črki zahteva multipleksorski pristop na izhodu avtomata.

Na sliki 3.16 vidimo tista modulska stanja, ki so kritična. Na levi strani slike vidimo razmere, ki so v originalnem drevesu na sliki 3.10 in na desni strani postopek prekinjanja zank. Oznake na desni strani slike pomenijo $0x_i = \bar{f}\, x_i$ in $1x_i = f\, x_i$. Modulsko stanje $\{2, 3\}$ moramo s funkcijo f ločiti



Slika 3.16: Izsek diagrama prehajanja stanj, ki je kritičen za povratno modularno shemo avtomata.

	$f=0$	$f=1$
$\{2,3\}$	$\{2\}$	$\{3\}$
$\{1,2\}$	$\{2\}$	$\{1\}$

Tabela 3.3: Ločitveni proces stanj avtomata.

v stanji $\{2\}$ in $\{3\}$ in stanje $\{1,2\}$ v stanji $\{1\}$ in $\{2\}$. Modulsko stanje $\{1,3\}$ nima direktne zanke, zato ga ločimo na $\{1\}$ in $\{3\}$ le, če že opravljene ločitve še niso odpravile zanke tudi za stanje $\{1,3\}$. Napravimo potrebni ločitvi stanj $\{2,3\}$ in $\{1,2\}$ tako, kot to določa tabela 3.3. Na tej osnovi pridemo do avtomata, ki je podan v tabeli prehajanja stanj 3.4 oziroma z diagramom prehajanja stanj na desni strani slike 3.16.

Oglejmo si sedaj generacijo sheme za funkcijo f. Vhod je demultipleksorsko razvejan s stopnjo $p=2$. Izhodni modul za funkcijo f ustreza stanju $\{1,3\}$. Predhodne module dobimo pri omenjeni razvejanosti vhodov na osnovi izraza (3.27) in tabele 3.4.

f	1	0	1
z	0	1	1
s_i	1	2	3
$\bar{f}\,x_1$	$-$	2	$-$
$f\,x_1$	2	$-$	3
$\bar{f}\,x_2$	$-$	1	$-$
$f\,x_2$	1	$-$	3
$\bar{f}\,x_3$	$-$	3	$-$
$f\,x_3$	2	$-$	1
$\bar{f}\,x_4$	$-$	3	$-$
$f\,x_4$	1	$-$	2

Tabela 3.4: Avtomat s prekinjenimi zankami pri kritičnih stanjih.

$$\begin{aligned} \mathrm{D}^1\{1,3\} = x_1\bar{f}\{-(1,3)\} \vee x_1 f\{3(2)\} \vee \\ \vee x_2\bar{f}\{2(1,3)\} \vee x_2 f\{1,3(2)\} \vee \\ \vee x_3\bar{f}\{2(1,3)\} \vee x_3 f\{3(2)\} \vee \\ \vee x_4\bar{f}\{2(1,3)\} \vee x_4 f\{1(2)\}\,. \end{aligned} \tag{3.27}$$

V oklepaju so navedena redundantna stanja, kar pomeni, da če se pojavijo, ni nič narobe. Simbol $-$ nam predstavlja podatek, da predhodno ni več nobenega stanja. Na novo so se pojavila stanja $\{3(2)\}$ in $\{1(2)\}$, katera moramo še opredeliti. Vsa ostala stanja, ki so:

- že opredeljena ali
- so v njih vsa stanja avtomata (opraviti imamo s popolno abecedo avtomata), sem šteje tudi množica $\{q(\text{vsi ostali v redundanci})\}$ ali
- predstavljajo prazno množico, sem šteje tudi $\{-(...)\}$,

jemljemo kot zaključena in tako ne večajo več razsežnosti sekvenčne sheme V. V (3.27) imamo tako polna stanja $\{2(1,3)\} \equiv 1$ in prazna stanja $\{-(1,3)\} \equiv 0$.

Nadaljevati moramo tako le še z modulskimi stanji $\{3(2)\}, \{1(2)\}$ in $\{1,3(2)\}$, za kar imamo:

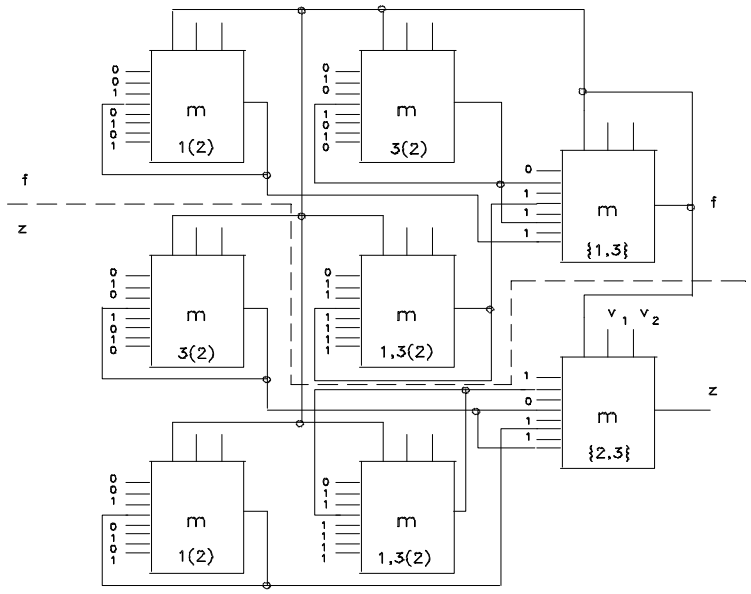
$$\begin{aligned}
\mathrm{D}^1\{3(2)\} = x_1\bar{f}\{-(1,2,3)\} \vee x_1 f\{3(1,2)\} \vee \\
\vee x_2\bar{f}\{-(2,3)\} \vee x_2 f\{3(2)\} \vee \\
\vee x_3\bar{f}\{2(1,3)\} \vee x_3 f\{-(1,2)\} \vee \\
\vee x_4\bar{f}\{2(1,3)\} \vee x_4 f\{-(2,3)\}
\end{aligned} \tag{3.28}$$

$$\begin{aligned}
\mathrm{D}^1\{1(2)\} = x_1\bar{f}\{-(1,2,3)\} \vee x_1 f\{-(1,2)\} \vee \\
\vee x_2\bar{f}\{2(1,3)\} \vee x_2 f\{1(2)\} \vee \\
\vee x_3\bar{f}\{-(1,3)\} \vee x_3 f\{3(1,2)\} \vee \\
\vee x_4\bar{f}\{-(1,3)\} \vee x_4 f\{1(2,3)\}\,.
\end{aligned} \tag{3.29}$$

$$\begin{aligned}
\mathrm{D}^1\{1,3(2)\} = x_1\bar{f}\{-(1,2,3)\} \vee x_1 f\{3(1,2)\} \vee \\
\vee x_2\bar{f}\{2(1,3)\} \vee x_2 f\{1,3(2)\} \vee \\
\vee x_3\bar{f}\{2(1,3)\} \vee x_3 f\{3(1,2)\} \vee \\
\vee x_4\bar{f}\{2(1,3)\} \vee x_4 f\{1(2,3)\}\,.
\end{aligned} \tag{3.30}$$

Na enak način pristopimo k izhodni funkciji oziroma črki z. Tej zadošča particija $P_a = \{\{1\},\{2,3\}\}$, kar pomeni, da je izhodni modul tisti, ki zadošča stanju oziroma modulu $\{2,3\}$

$$\begin{aligned}
\mathrm{D}^1\{2,3\} = x_1\bar{f}\{2(1,3)\} \vee x_1 f\{1,3(2)\} \vee \\
\vee x_2\bar{f}\{-(1,3)\} \vee x_2 f\{3(2)\} \vee \\
\vee x_3\bar{f}\{2(1,3)\} \vee x_3 f\{1(2)\} \vee \\
\vee x_4\bar{f}\{2(1,3)\} \vee x_4 f\{3(2)\}\,.
\end{aligned} \tag{3.31}$$



Slika 3.17: Modularna shema s povratno funkcijo f za avtomat v tabeli 3.4.

Vključevanje modulov $\{3(2)\}, \{1(2)\}$ in $\{1, 3(2)\}$ v (3.31) smo pravzaprav že poznali v okviru izrazov (3.28) do (3.30) pri določanju sheme za f, zato nam teh treh ni potrebno ponavljati pri računanju sheme za z. Naj še posebej omenimo, da množica $\{1, 3(2)\}$ ni element množice M^*, saj še nima napravljene selekcije med stanjema 1 in 3.

Tako. Sedaj imamo opredeljene vse potrebne povezave med vsemi moduli in zato lahko narišemo povratno modularno shemo za opazovani avtomat. Takšno shemo vidimo na sliki 3.17.

V gornjem besedilu smo uporabljali pojem modulsko stanje, ko smo opazovali določen modul, zato da smo lahko opisali postopek določanja shem V_f in V_z. Poleg avtomatnega stanja in modulskega stanja imamo še stanje modulske sheme, ki je predstavljeno z dvojiško kodo izhodov vseh modulov v modularni shemi. Vemo namreč, da je izhod vsakega modula v shemi po vrednosti ali 1 ali 0.

Čeprav obstaja praktična enakost med podstrukturama V_f in V_z (na sliki 3.17 sta ločeni s črtkano črto), te značilnosti ne moremo izkoriščati v smislu minimizacije modularne sheme. Zavedati se moramo, da sta V_f in

V_z med seboj neodvisni strukturi in se zato lahko ena nahaja v stanju S_f in druga v S_z, pri čemer je $S_f \neq S_z$. Glede na to je, v splošnem gledano, delovna sekvenca v V_f lahko povsem drugačna od delovne sekvence v V_z.

3.6 NAČINI DELOVANJA LOGIČNEGA MODULA

Pri načrtovanju RSS se dogaja, da imamo sicer v vhodni bazi podatkov mnogo najrazličnejših modulov, da pa pri določenem načrtovalnem detajlu ni niti enega, ki bi neposredno ustrezal. V takšnem primeru lahko napravimo nov modul in ga vstavimo v bazo modulov, tudi za kasnejšo uporabo. Tak pristop je zahteven, dolgotrajen in vprašanje je, če kot načrtovalec RSS sploh znamo ažurirati bazo modulov. Glede na ta in podobne probleme v dobrih orodjih ne ponudijo le modul kot tak, temveč tudi vse načine delovanja modula. Če ima modul m vhodov, je na vsakem takšnem vhodu lahko vrednost 0 ali 1 ali spremenljivka x. Vseh možnosti ni le 2^m, temveč 3^m. Če pri našem detajlu načrtovanja nastopa n spremenljivk, je lahko na kateremkoli vhodu modula katerkoli spremenljivka $x_i, i = 1, 2, ..., n$ ali v obliki $\bar{x}_i$ ali v obliki x_i. Pri vsaki izbiri spremenljivk in konstant na vhodu modula lahko dobimo drugačen izhod oziroma stalni modul lahko opravlja vrsto logičnih funkcij, med katerimi so nekatere lahko tudi enake med seboj. V opisanem primeru pravimo, da ima stalni modul (v njega ne posegamo) različne načine delovanja.

Načine delovanja modula lahko računamo na osnovi njegovega opisnega vektorja $O = (V_0, V_1, I_0, I_1)$. Pri tem je V_0 število vhodov, ki so negirani (0-vhodov), V_1 število vhodov, ki so nenegirani (1-vhodov), I_0 število negiranih izhodov (0-izhodov) in I_1 število nenegiranih izhodov (1-izhodov).

Logične funkcije so v sodobnih RSS običajno realizirane na osnovi NAND- in NOR-operatorjev. Čeprav je temu tako, načine delovanja opazujemo z disjunktivnega in konjuktivnega vidika, ker postulati algebre logike vendarle glasijo na $\{\vee, \&, ^- \}$. Tako se pri verigi med seboj povezanih NAND operatorjih načini delovanja menjajo v smislu

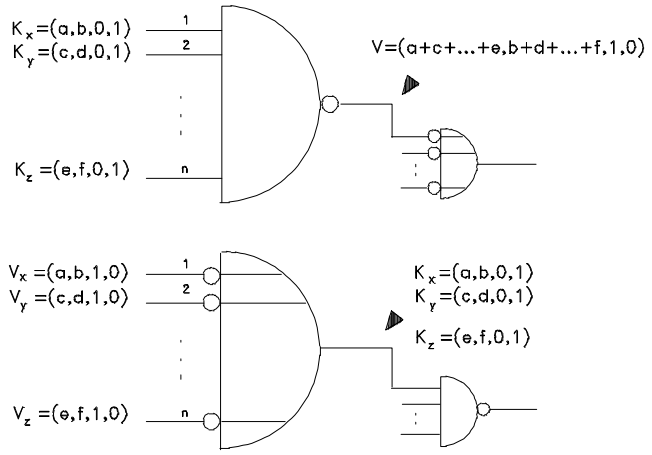
disjunktivni n. $\rightarrow$ *konjunktivni n.* $\rightarrow$ *disjunktivni n.* $\rightarrow$ *konjunktivni n.* ...

ali

konjunktivni n. $\rightarrow$ *disjunktivni n.* $\rightarrow$ *konjunktivni n.* $\rightarrow$ *disjunktivni n.* ... ,

ker vsebuje NAND poleg konjunkcije tudi negacijo le-te. Kako prehajamo

z načini delovanja preko operatorja NAND vidimo na sliki 3.18. V gornjem delu slike vidimo, kao dobimo iz konjunktivnih (K-) načinov disjunktivne (D-) načine delovanja. V spodnjem delu slike je podan obraten primer: iz D-načinov dobimo K-načine delovanja operatorja NAND.



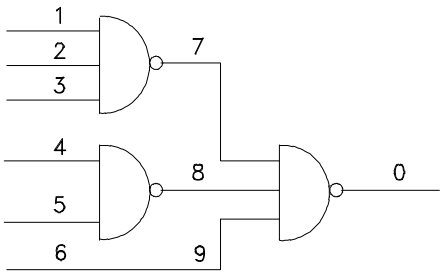
Slika 3.18: Prehajanje iz K-načina na D-način delovanja pri operatorju NAND.

Značilnost slike 3.18 je v tem, da je nenegirani vhod lahko vezan le na nenegiran vhod in negiran izhod samo na negiran vhod naslednjega operatorja.

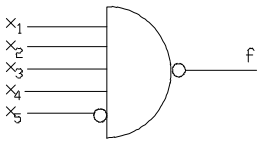
Logična shema se začenja z vhodom oziroma 'žico', ne pa z operatorjem. Opisni vektor tega pričetka logične sheme je $O = (0, 1, 0, 1)$ ali $O = (1, 0, 1, 0)$ odvisno od tega, ali začenjamo shemo postavljati z negiranim ali nenegiranim vhodom prvega operatorja v shemi.

Toliko o opredelitvi K- in D-načina delovanja. Vzemimo sedaj modul na sliki 3.19 in poskusimo ugotoviti, koliko K-načinov in, koliko D-načinov delovanja ima. Vsak vhod vseh operatorjev v modulu je označen s svojim imenom. Princip računanja po sliki 3.18 nam da naslednje načine delovanja:

$$
\begin{aligned}
d_1 &= D_{0-7,1-8,4-9,6} = (2, 1, 1, 0) \\
d_2 &= D_{0-7,1-8,5-9,6} = (2, 1, 1, 0) \\
d_3 &= D_{0-7,2-8,4-9,6} = (2, 1, 1, 0) \\
d_4 &= D_{0-7,2-8,5-9,6} = (2, 1, 1, 0) \\
d_5 &= D_{0-7,3-8,4-9,6} = (2, 1, 1, 0)
\end{aligned}
$$



Slika 3.19: Modul z označbo, ki jo potrebuje analiza njegovih načinov delovanja.



Slika 3.20: Logično vezje z opisnim vektorjem $O = (1, 4, 1, 0)$.

$$\begin{aligned} d_6 &= D_{0-7,3-8,4-9,6} = (2, 1, 1, 0) \\ k_1 &= K_{0/7-1,2,3} = (0, 3, 0, 1) \\ k_2 &= K_{0/8-4,5} = (0, 2, 0, 1) \\ k_3 &= K_{0/9-9,6} = (1, 0, 0, 1)\,. \end{aligned} \tag{3.32}$$

Te načine lahko določimo numerično po sliki 3.18 ali pa jih določimo na osnovi MDNO in MKNO izhodne funkcije modula na sliki 3.19 [9].

Vzemimo, da moramo z modulom na sliki 3.19 realizirati preprosto logično vezje na sliki 3.20, ki ima opisni vektor $O = (1, 4, 1, 0)$. Če z načini delovanja (3.32) lahko pokrijemo delovanje sheme na sliki 3.20, je problem rešljiv sicer pa ne. Kako zgleda numerični postopek, ki nam je v pomoč pri takšnem ali podobnem pokrivanju?

Očitno je, da z enim modulom ne moremo zadostiti shemi na sliki 3.20. Zato gremo v večje število zaporedno vezanih modulov s ciljem, da nam veriga le-teh ob kombinaciji različnih načinov da ustrezni rezultat. Za vsak način izberemo selekcijsko spremenljivko, ki se nanaša na njegovo prisotnost oziroma odsotnost v shemi. Tako vzamemo

$$\begin{aligned} y_i &= 1 \text{ , če je } i\text{-način prisoten v shemi} \\ y_i &= 0 \text{ , če je } i\text{-način odsoten v shemi.} \end{aligned} \tag{3.33}$$

Rešitev nam daje sistem neenačb

$$\begin{aligned} &\sum_{i=1}^{n} V_0(i)\, y_i - \sum_{i=1}^{n} I_0(i)\, y_i \geq V_{v0} - I_0(1) \\ &\sum_{i=1}^{n} V_1(i)\, y_i - \sum_{i=1}^{n} I_1(i)\, y_i \geq V_{v1} - I_1(0) \\ &\sum_{i=1}^{n} I_0(i)\, V_1(i)\, y_i \geq 1 \\ &\sum_{i=1}^{n} I_1(i)\, V_0(i)\, y_i \geq 1\,. \end{aligned} \tag{3.34}$$

Pri tem je n število serijsko vezanih modulov z opisnimi vektorji, i je tekoči indeks modula s opisnim vektorjem $O(i) = (V_0(i), V_1(i), I_0(i), I_1(i))$ in $(V_{v0}, V_{v1}, I_{v0}, I_{v1})$ je opisni vektor ciljnih vrat (v primeru slike 3.20 je to O_v = (1,4,1,0)). Zadnja dva izraza v (3.34) sta le pogoja, da če ostajajo nenegirani (negirani) vhodi, morajo biti na voljo tudi nenegirani (negirani) izhodi, ker lahko povezujemo le I_0 z V_0 in I_1 z V_1. $I_0(1)$ in $I_1(0)$ so konstante, ki izhajajo iz tega, da se shema drevesno razceplja od enega izhoda proti večjem številu vhodov. Če ciljna shema ne bi bila vrata z enim izhodom, postavki $I_0(1)$ in $I_1(0)$ ne bi bili konstanti.

Postavimo sedaj sistem neenačb za vrata na sliki 3.20. Najprej priredimo spremenljivke y načinom modula na sliki 3.19. Vzemimo:

$$\begin{aligned} y_1 &: d_1 = (2,1,1,0) \\ y_2 &: d_2 = (2,1,1,0) \\ y_3 &: d_3 = (2,1,1,0) \\ y_4 &: d_4 = (2,1,1,0) \\ y_5 &: d_5 = (2,1,1,0) \\ y_6 &: d_6 = (2,1,1,0) \\ \\ y_7 &: k_1 = (0,3,0,1) \\ y_8 &: k_2 = (0,2,0,1) \\ y_9 &: k_3 = (1,0,0,1)\,. \end{aligned}$$

Spremenljivke $y_1 - y_6$ pokrivajo disjunktivne načine $d_1 - d_7$, zadnje tri, $y_7 - y_9$ pokrivajo konjunktivne načine $k_1 - k_3$. Te prireditve vodijo k sistemu neenačb

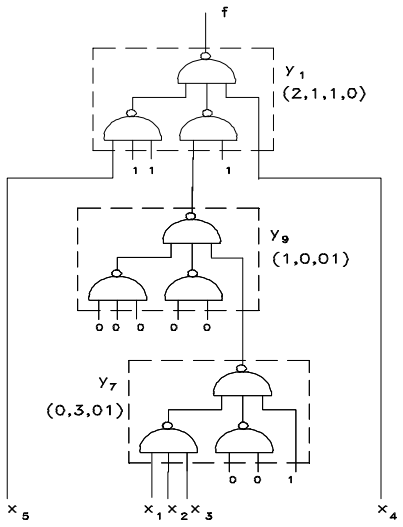
$$2y_1 + 2y_2 + 2y_3 + 2y_4 + 2y_5 + 2y_6 - y_1 - y_2 - y_3 - y_4 - \\ -y_5 - y_6 - y_9 \geq 1 - 1$$

$$y_1 + y_2 + y_3 + y_4 + y_5 + y_6 + 3y_7 + 2y_8 + y_9 - x_7 - x_8 \geq 4.$$

Očividno je, da imamo veliko rešitev že zaradi tega, ker je veliko enakih disjunktivnih načinov delovanja. Ena od teh rešitev je

$$y_1 = y_7 = y_9 = 1$$
$$y_2 = y_3 = y_4 = y_5 = y_6 = y_7 = y_8 = 0.$$

V naši modularni shemi imamo tako tri načine



Slika 3.21: Modularna shema za vrata na sliki 3.20.

$$y_1 : d_1 = (2, 1, 1, 0)$$
$$y_7 : k_2 = (0, 3, 0, 1)$$
$$y_9 : k_3 = (1, 0, 0, 1)\,.$$

Potrebna je še kontrola logičnih pogojev, to je zadnjih dveh izrazov v (3.34). Lahko jih uporabimo samo v primeru y_1, y_7 in y_9, ker so vsi ostali $y_i = 0$. Imamo

$$1 \cdot 1 + 3 \cdot 0 + 0 \cdot 0 = 1 \geq 0$$
$$2 \cdot 0 + 0 \cdot 1 + 1 \cdot 1 \geq 0\,.$$

Torej ni nobenih težav pri tipih vhodov in izhodov. Modularno shemo za vrata na sliki 3.20 vidimo na sliki 3.21. Neuporabljeni vhodi so vezani na konstante 0 in 1 tako, da je na izhodu funkcija f, kot to zahtevajo omenjena vrata.

3.7 PREHOD NA CELIČNE RSS

Do sedaj smo vseskozi govorili o modulu. Zanimalo nas je, kako uvesti čim bolj enake logične podsestave in kako te povezati med seboj, da pridemo s tem do modularne sheme, ki predstavlja načrtovano RSS. Modularnost se je začela razvijati s pojavom tiskanin, v večji meri pa se je razmahnila takoj, ko se je pojavila mikroelektronika s SSI-čipi. Čip je imel v začetku majhno stopnjo integracije, tako da sam kot tak ni mogel pokriti velike logične vsebine. Logično shemo so tedaj postavljali z večjim številom čipov. S stopnjo integracije je logična vsebina (število ekvivalentnih vrat) na čip hitro rasla, tako da se je v en čip dalo vstaviti že kar velike logične sheme. Vendar, s tem ko je rasla stopnja integracije, je rasla tudi potreba po še večji logični vsebini. V času ko so na en čip postavljali kompleten 4-bitni mikroprocesor, so že razvijali 8-bitnega. Ko so tega postavili na čip, so že načrtovali 16-bitni mikroprocesor, za tem 32-bitnega itd. Podobno je bilo tudi pri drugih RSS in ne samo pri procesorjih. Čim bolj se je čip s svojo logično vsebino povečeval, tem bolj je izgubljal pomen preprosti logični modul. Na čip so postavljali vedno bolj celovite RSS z lastno aplikacijsko težnjo in razsežnostjo. Že v dobi srednje integracije (doba PAL-ov) se je začel uveljavljati nov pojem, celica, ki je prevzel vlogo modula. Danes poznamo celice z najrazličnejšimi pridevniki (vhodna celica, izhodna celica, standardna celica, makrocelica, reprogramirna celica, megacelica itd.). Celica je v tem času tipizirana enota (lahko element, "device"), ki je lepo tehnološko in logično zaokrožena v dokaj uporabno enoto. Tipizirana in uporabna enota pomeni, da je njen proizvodni proces optimiziran, frekvenca uporabe celice na čipih pa zavisi od logične univerzalnosti ali vsaj smiselnosti, ki naj ne bi bila

nizka. V tem času so predvsem na pohodu ASIC-načrtovanja (application-specific integrated circuit design), ki dajo na tržišče ASIC-elemente. Poznana je na primer tehnika CBIC (cell-based integrated circuits), v kateri se množično uvajajo standarne celice, ki so predhodno načrtane.

Logični modul pa je začetnik še enega, danes množično uporabljenega pojma v RSS, to je logične razporeditve (gate arrays, GA). Pri teh gre za množično ceneno replikacijo osnovnih celic (base cells). Cenenost pri tem pomeni, da sta pri proizvodnji potrebni samo dve matalni maski. Nasproti standarnim celicam se ne poceni samo proizvodnost, temveč tudi načrtovanje. Če je čas, ki je potreben za načrtovanje standardne celice, 6 mesecev, je za načrovanje GA potrebno le 6 tednov. Najdražji elementi so ASIC-elementi (full-custom devices); uveljavljajo se predvsem zaradi največje stopnje integracije. Vsi drugi, kot so: standardne celice, GA, PLD (programmable logic device) itd. so cenejši, čas načrtovanja je krajši, manjša pa je, žal, tudi stopnja integracije. Poprečni čas načrtovanja polnega ASIC-elementa je 9 mesecev, torej občutno večji od ostalih.

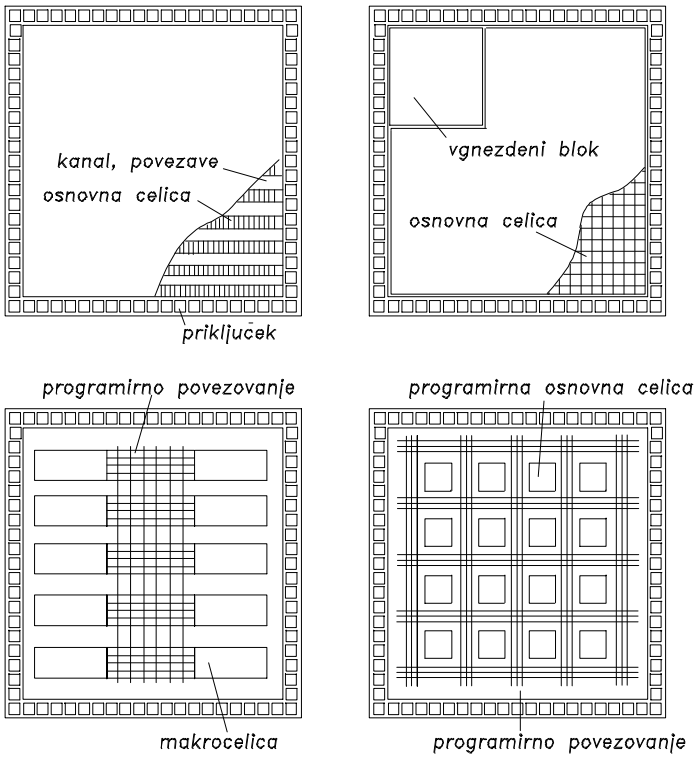
Naj na tem mestu v osnovi razčistimo tudi kratice PAL, PLA, PLD, EPLD, FPLD in FPGA. P v teh imenih je edina črka, ki ne dela razlik med vsemi štirimi navedenimi elementi oziroma tehnikami. P pomeni, da gre za programirni gradnik. Proizvajalec takšnega gradnika da kasnejšemu načrtovalcu na voljo programirnost, da ta s svojim programiranjem zaključi že prej proizvedeno delno funkcionalnost gradnika. Najstarejša programirna tehnika je programirna razporejena logika PAL (programmable array logic). Pri PAL je uporabljena ista tehnika, ki se je uveljavila pri programirnih bralnih pomnilnikih PROM (mask-programmed ROM) z namenom, da se dosežena gibljivost pri PROM uporabi tudi pri logičnih shemah (začetki tega procesa sodijo v ovir firm Monolithic Memories, AMD). Rezultat tega pristopa so bila PAL-vezja. Rekli smo "bili", čeprav se PAL-čipi še danes tržijo in uporabljajo. Če gremo s principi programirnosti bralnih pomnilnikov na ASIC-concept razporejanja (custom ASIC), pridemo do PLA elemantov oziroma tehnike. Bistvena razlika med PAL in PLA je, da ima PAL le AND-programirnost, medtem ko ima, novejši, PLA poleg AND-programirnosti tudi OR-programirnost. PAL ima OR-matriko stalno, torej zaprto za kasnejšega načrtovalca. Če smo gradnik že enkrat programirali, mu vsebino lahko brišemo in ga ponovno programiramo. V tem primeru imamo na primer za PLD kratico EPLD. Realizacije zgodnjih PAL-, PLA- in PLD-gradnikov so postavljene z bipolarno tehnologijo in ustreznimi programirnimi upori (fuses). Danes je že poznana CMOS PLD tehnika, v kateri imamo namesto fuse-programirnih uporov (tranzistorjev)

"floating-gate tranzistorje".

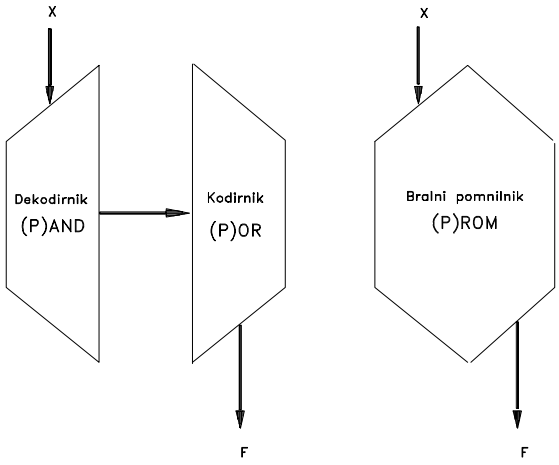
FPGA (field-programmable gate array) je na nek način PLD z razliko, da gre pri FPGA za večjo kompleksnost tehnologije in večjo logično vsebino. Vsi zgoraj omenjeni primerki tehnik na nek način sodijo v ASIC-okvir, s tem da je FPGA najnoveši element te družine elementov. Na sliki 3.22 vidimo skice štirih ASIC-čipov. Levo zgoraj je GA-gradnik. Med vrstami osnovnih celic je prostor za povezovanje celic, zato je to kanalski tip GA. Obstajajo tudi GA brez kanalov (sea-of-gates, SOG). Na desni zgoraj je GA-gradnik oziroma čip z vgnezdenim blokom. Tak blok je na primer lahko statični pomnilnik. Vso ostalo površino zasadajo osnovne logične celice. Levo spodaj je PLD-gradnik. Makrocelice so med seboj povezane z relativno obsežnim povezovalnim blokom. Logičnim makrocelicam večkrat sledijo pomnilne celice, tako da so PLD blizu avtomatnim realizacijam. Desno spodaj vidimo FPGA-gradnik. Če primerjamo oba spodnja gradnika, vidimo, da je pristop za povezovanje celic v večje strukture dokaj različen. Pri FPGA so povezovalne možnosti okoli vsake osnovne celice. Programirne so tako povezave kot tudi same celice. Pri PLD so programirne samo povezave.

Omenili smo že AND- in OR-programirnost. Kaj te programirnosti pomenijo? Omenili smo tudi, da je do PAL-čipov prišlo tako, da so progamirne možnosti, ki so se uveljavile pri pomnilnikih (prehod iz ROM na PROM, EPROM) prenesli tudi v okvir načrtovanja logičnih funkcij. ROM pomni informacije, ki smo jih predhodno vstavili vanj. Če te informacije glasijo na logične funkcije, opazovani ROM opravlja vlogo realizacije (emulacije) logičnih funkcij. Na enem izhodnem kanalu pomnilnika dobimo funkcijo $f(x_1, x_2, ..., x_n)$, če ima pomnilnik dvojiško naslavljanje s kodami dolžine n. Vsak naslov igra vlogo enega minterma v PDNO funkcije $f(x_1, x_2, ..., x_n)$. Na sliki 3.23 vidimo narisan bralni princip realizacije omenjene logične funkcije. Vektor naslavljanja $\mathbf{x} = (x_1, x_2, ..., x_n)$ predstavlja neodvisne spremenljivke, medtem ko izhodni vektor $\mathbf{F} = (f_1, f_2, ..., f_m)$ daje m neodvisnih logičnih funkcij. Če so te funkcije zapisane v PDNO, so realizirane v prvem logičnem nivoju s konjunkcijami (dekodirna matrika) in v drugem z disjunkcijami (kodirna matrika). Če obe omenjeni matriki postavimo skupaj, dobimo ravno gradnik ROM. (P) pomeni, da matriki lahko programiramo oziroma ROM je v tem primeru programirne narave. PAND in OR nam dasta značilnost za PAL, medtemo ko nam PAND in POR dasta značilnost za PLA.

Poglejmo si na primer 8-bitni EPROM MCM68764. Ima 8192 8-bitnih besed in nam zato glede na sliko 3.23 naenkrat lahko realizira 8 logičnih



Slika 3.22: Sodobni ASIC-gradniki.



Slika 3.23: Bralni princip relizacije logične funkcije.

funkcij. Da ima PROM lahko 8192 dosegljivih besed, ima 12-bitno naslavljanje. Torej gre za vhodni vektor $\mathbf{x} = (x_1, x_2, ..., x_{12})$, ki vsaki od osmih funkcij dodeli $2^{12} - 1 = 8191$ mintermov, kar je za praktično rabo odločno preveč. V poprečju načrtovanja logičnih funkcij bi bilo pri $n = 12$ dovolj že okrog 200 mintermov. Iz tega lahko potegnemo zaključek, da je velikost bralnih pomnilnikov neprilagojena za načrtovanje logičnih funkcij (namenjena je pomnjenju ne pa decizijskim funkcijam), medtem ko je tehnika programiranja za to načrtovanje povsem na mestu. Pri prehodu iz PROM na PAL oziroma PLA obdržimo le manjši del naslovov, prihranjeni prostor na čipu pa zapolnimo s pomnilnimi celicami ali s čem drugim, kar je dobro, da je v bližini logičnih funkcij $f(x_1, x_2, ..., x_n)$. Pomnilnik MCM68764 ima $8 \times 8192 = 65536$ bitnih mest, kar znese 2^{65536}=10^{19728} stanj. Ta postavka je z vidika realizacije logičnih funkcij na osnovi logične površine (gate array) ogromna postavka. V primeru realno uporabnih funkcij potrebujemo mnogo manj. Že zaradi tega velikostni red bralnih pomnilnikov ni primeren za načrtovanje: od bralnih pomnilnikov vzamemo le tehnologijo in tehniko programiranja, medtem ko dimenzije priredimo za načrtovanje.

Če se še enkrat ozremo na strukture na sliki 3.22, lahko zaključimo, da so te še vedno modularne sheme. Sofisticirane logične module imenujemo logične celice, te pa tudi fizično med seboj povezujemo na sofisticiran način, ki ga pred ASIC-tehniko nismo poznali. Predvsem se je v sofisticiranost

celic in povezav med njimi vsilil pojem programirnosti (programabilnosti) (to be programmable), ki načrtovanje RSS postavlja na dosti višji nivo, kot je to bilo na primer pred 20 leti. V naslednjih poglavjih bomo v večji meri obravnavali prav problematiko, ki smo jo načeli v tem razdelku.

Bralca najbrž zanima, kje se konča mikroelektronsko in prične računalniško načrtovanje? Mikroelektronsko načrtovanja se nanaša na načrtovanje čipov, ker so čipi proizvodi mikroelektronike. Stvar računalnikarja pa je, da zna implementirati na tržišču dosegljive čipe v veččipni logični shemi. Posebej pride računalniško (ne mikroelektronsko) načrtovanje do izraza, če so čipi programirni. V tem primeru računalnikar preko programirnosti čipe lažje prilagojuje aplikaciji.

3.8 RAČUNALNIŠKA POMAGALA PRI NAČRTOVANJU RSS

V prvem delu smo spoznali vrsto postopkov, ki so primerni za fizično in logično načrtovanje oziroma postavljanje RSS. Osnovna naloga načrtovalca RSS je, da zna omenjene in podobne postopke matematizirati do te mere, da lahko pristopi k njihovemu programiranju in iz programskih podsestavov sestavi CAD (computer aided design) programsko orodje za načrtovanje RSS. Pravzaprav je to naloga načrtovalca - metodologa, medtem ko je naloga vsakega načrtovalca, da zna že postavljena orodja za RSS koristiti v vsakdanji praksi. Kot v splošnem CAD imamo tudi v primeru načrtovanja RSS orodja, ki so profesionalna in draga ter profesionalno manj sposobna, ki pa so zato cenena in računalniško nezahtevna. Od CAD za RSS pričakujemo, da ta vsebujejo

- postavljanje modulov na kartice, povezovanje modulov na osnovi tiskanin, preverjanje napak in izdelavi proizvodne dokumentacije večstranskih kartic
- bogato izbiranje vnaprej pripravljenih komponent (vhodne baze različnih komponent)
- vnašanje logičnih shem v računalnik (ASCII, simboli, diagram prehajanja stanj)
- logično minimizacijo shem, logično računanje, logične optimizacije
- grafično urejanje shem

- hierarhično načrovanje logičnih shem
- logično simulacijo, testiranje, popravljanje logičnih napak
- izdelavo standardizirane dokumentacije logičnih shem (načrti, sestavnice komponent, spisek redundance, reporti, programirne tabele)
- časovno analizo logičnih shem (logični čas)
- časovno analzo logičnih shem z vidika elektronike (realni čas)
- statistiko, ekonomsko analizo načrtovanja in še kaj.

Boljša orodja za omenjeno delo imajo značilnost opisnega jezika za aparaturno opremo (HDL, hardware description language). Tak jezik naj bi imel prevajalnik, grafični urejevalnik in dober vhodno-izhodni jezik. Med najbolj razvpita CAD orodja za RSS sodi VHDL (Very high-speed IC HDL), ki ga je v zgodnjih 80 letih forsiral DoD (U.S. Department of Defence). Tega je privzel tudi IEEE pri izdelavi standardov 1076-1987, v okviru DoD pa je bil uporabljen kot verifikacijski medij pri ASIC-standardih MIL-STD-454. Temu se želi približati vrsta drugih orodij, ki kažejo večjo uporabnost morda samo za določene namene. Tako je eno orodje zelo uporabno za logično načrtovanje, drugo samo za PAL-tehniko tretje samo za tiskanine itd. Poznana orodja so:

- ABEL-HDL (Advanced Boolean Expression Language), uporaben pri načrtovanju s PAL, PLD, FPGA
- CUPL (Universal Compiler for Programmable Logic), uporaben pri načrtovanju s PLD,
- LOG/iC, uporaben pri načrtovanju s PLD
- PALASM (PAL Logic Equation Assembler), uporaben predvsem pri načrtovanju s PAL, PLD
- SYNARIO (Universal FPGA Design System), uporaben pri načrtovanju FPGA, PLD
- CADES-G (Computer Aided Design and Engineering System Graphic), načrtovanje PCB
- PLA Tools, uporaben pri načrtovanju s PLA

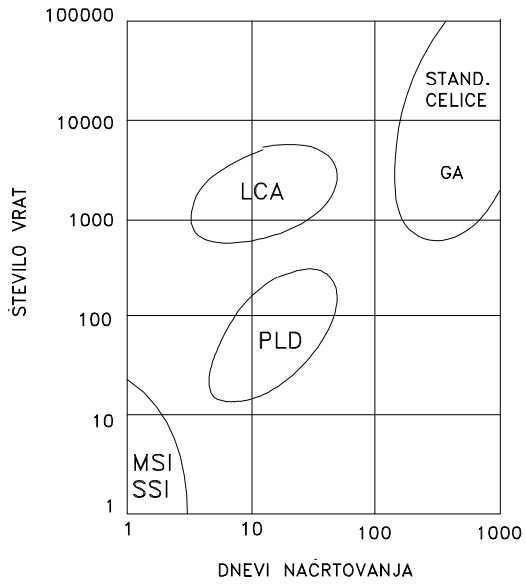
- XACT (Xilinx Development System), načrtovanje LCA (logic cell array), CLB (user configurable logic blocks), IOB (programmable input-output blocks)
- A+PLUS (Altera Programmable Logic User System), EPLD
- Electronics Workbench, uporaben pri načrtovanju z elektronskimi in IC-vezji (s tem orodjem je v tem učbeniku narisanih kar nekaj logičnih shem)
- PCAD, načrtovanje elektronskih vezij, PCB.

V zadnjem času imamo za splošno osnovno načrtovanje RSS v naših načrtovalskih laboratorijih največ opraviti z orodjem P-CAD [8], ki teče pod MS DOS. To orodje prej sodi v elektroniko kot na področje načrtovanja RSS. Ostaja na nivoju vrat, osnovnih IC-vezij in daje malo dodatne pomoči za kakršnokoli posebno tehniko, ki se uveljavlja pri načrtovanju RSS. Orodje deluje nad komponentami:

· komponente AMD,
· komponente CMOS,
· komponente TTL,
· diskretne komponente,
· nehomogene komponente DSP,
· komponente ECL,
· elektromehanične komponente,
· mikroprocesorji INTEL,
· linearne komponente,
· komponente Zilog,
· mikropocesorji MOTOROLA,
· komponente NS.

Pri načrtovanju so nam na voljo razna splošna povezovalna pomagala na druga orodja, kot na primer:

· PSPICE (simulacije vezij)
· PC-XILINX (vmesnik k orodju XACT)
· PCAM (proizvodni podatki na osnovi Gerber datotek)
· PCDXF (prehod iz P-CAD na AutoCAD, MIKRO CADAM translator)
· PC-GPLOT (Gerber datoteke za risanje na HP74XX, 75XX, 76XX, HI-DMP, HP LJ, Laser Master, PostScript).



Slika 3.24: Doba načrtovanja s programirnimi gradniki MSS, PLD, LCA, GA, STANDARDNE CELICE.

Žal je v tem spisku za računalnikarje zanimiv le vmesnik k XACT. Želeli bi si takšne vmesnike tudi k drugim navedenim orodjem za načrtovanje RSS.

V tem delu nimamo namena celovito opisovati orodij za načrtovanje RSS. Sem pa tja bomo izkoristili le vhodni-izhodni jezik kakšnega orodja sicer bi delo postalo preobsežno.

V kolikor smo z omenjenimi (in ne omenjenimi) potrebnimi orodji založeni, imamo pri načrtovanju opraviti z diagramom na sliki 3.24. V njem vidimo odvisnost med številom ekvivalntnih vrat RSS in številom dni ustreznega načrtovanja. Naj poudarimo, da je univerzalnejše tisto orodje, ki ima več vmesnikov ozioma filtrov za druga morda specialnejša orodja.

Del II

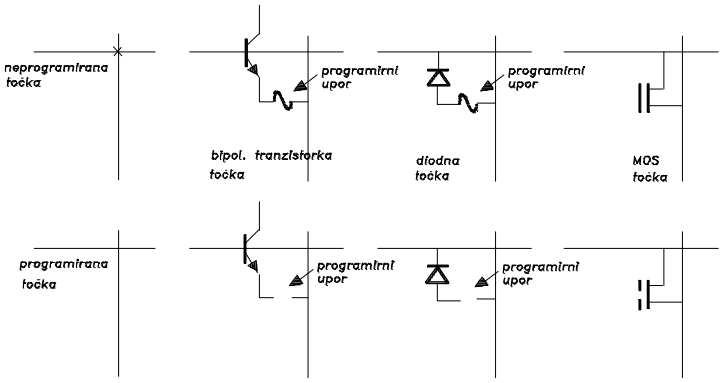
PROGRAMIRNOST IC-GRADNIKOV RSS

Poglavje 4

PROGRAMIRNOST PAL-GRADNIKOV

4.1 PRINCIP PAL-PROGRAMIRANJA

PAL-tehnika ima že petnajstletnico, kar pomeni, da sodi med najstarajše programirne tehnike. Družino, v katero sodi, sestavljajo PAL, FPLA in PROM. PAL gradniki so bili izredno koristni, ker so se učinkovito implementirali v logične sheme, pri katerih v realizacijo sploh še ni bila vpeljana pomembnejša integracija. S PAL tehniko so se razmere krepko spremenile. V začetku je bila najprej HAL-tehnika (hardware array logic), pri kateri so načrtovalci pošiljali preproste perforirane papirne kartice k proizvajalcu, ki je poslane podatke 'zapekel' na čip, katerega je kasneje dostavil načrtovalcu. Programirni upor (fuse) je ta pristop zelo poenostavil, saj je omogočil, da je načrtovalec lahko doprogramiral čip kar v domačem laboratoriju. PAL-realizacije so zelo zmanjšale velikost kartic oziroma na eno kartico je načrtovalec lahko postavil precej več logične vsebine kot v primeru neintegrirane logike in SSI. PAL-gradniki so prinesli s seboj zaščitno tehniko, ki je na aparaturni način resnično preprečila kopiranje (kraja podatkov). PAL-čipe se je dalo (se še vedno da) programirati na programatorjih za PROM. Množičnost uporabe PAL-gradnikov je bila in je še velika, zato se je tudi močno razvila ustrezna tehnika programiranja. Postavili so standard JEDEC JC-42.1-81-62 (The Joint Electron Devices Engineering Consil), s formatom PLDF (programmable logic data transfer format), ki je poenotil na programatorjih izhodne podatke za neposredno programiranje PAL-čipa. Ta se ob programiranju nahaja v podstavku, ki je izhodna



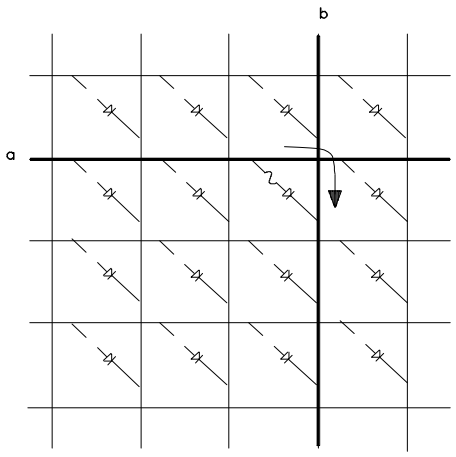
Slika 4.1: Programirna točka za PROM- in PAL-programiranje.

naprava računalnika.

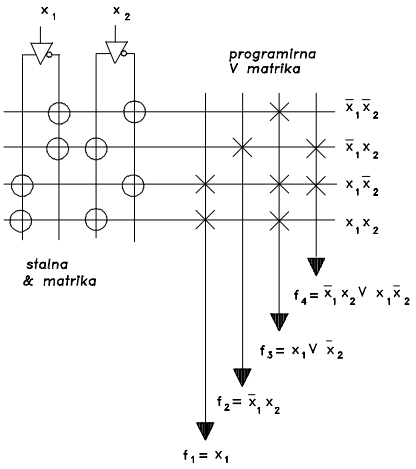
Osrednji objekt pri programiranju je programirni upor (fuse). Na sliki 4.1 vidimo programirno točko v treh izvedbah: z bipolarnim tranzistorjem, diodo in MOS-tranzistorjem. Zgornja vrsta predstavlja neprogramirano (nespremenjeno, "intact", "unblown") stanje, spodnja pa programirano (spremenjeno, "blown") stanje. V prvi vrsti je ohmski upor programirnega upora 0 ohmov in v drugi ∞ ohmov. Seveda so to idealne vrednosti, v praksi so lahko nekaj ohmov oziroma nekaj mohmov. Pri proizvodnji gradnika sta navpičnica in vodoravnica spojeni, po programiranju na stanje 'spremenjeno' načrtovalec prekine spojenost.

Programirne točke na sliki 4.1 so navadno postavljene v matriko. Na sliki 4.2 imamo matriko velikosti 4×4. V tej so vse programirne točke programirane, razen točke na preseku a in b. Zato lahko podatke pošiljamo iz horizontale a ne vertikalo b.

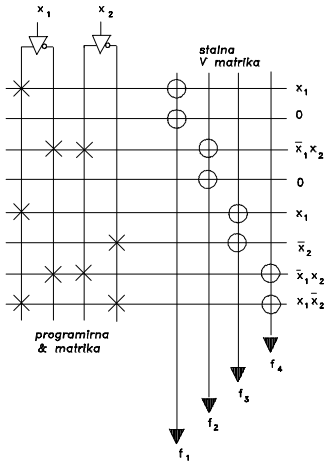
Že na sliki 3.23 smo ugotovili, da je PROM sestavljen iz dekodirne in kodirne matrike. Prva nam da vse potrebne konjunkcije in druga vse potrebne disjunkcije. Pri PROM se da programirati kodirno matriko, medtem ko je dekodirna matrika za načrtovalca RSS stalna (na sliki 4.3 so ustrezne točke označena z $\circ$), saj je dokončno postavljena že, ko je čip izdelan. Vodoravne črte nam vedno predstavljajo konjunktivno polje in navpične črte disjunktivno polje povezav. Izhodi na dekodirniku so mintermi logičnih funkcij, medtem ko izhodi kodirnika dajo disjunktivno združevanje mintermov. Tako imamo na sliki 4.3 funkcije



Slika 4.2: Matrika programirnih uporov.



Slika 4.3: Princip PROM-programiranja logičnih funkcij.



Slika 4.4: PAL-princip programiranja logičnih funkcij.

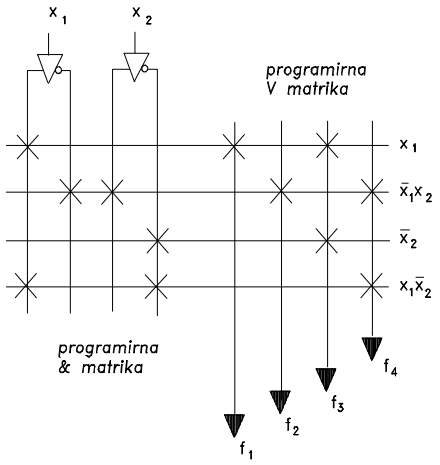
$$
\begin{aligned}
f_1 &= x_1x_2 \vee x_1\bar{x}_2 = x_1 \\
f_2 &= \bar{x}_1x_2 \\
f_3 &= \bar{x}_1\bar{x}_2 \vee x_1\bar{x}_2 \vee x_1x_2 = x_1 \vee \bar{x}_2 \\
f_4 &= x_1\bar{x}_2 \vee \bar{x}_1x_2.
\end{aligned}
\tag{4.1}
$$

Slika 4.4 je tipična za PAL-programiranje. Na tej sliki je postala stalna kodirna matrika, medtem ko je dekodirna postala programirna. Vodoravne linije niso več samo mintermi, temveč je takšna linija kakršnakoli konjunkcija, vključno tudi konstanta 0 ali 1. Seveda je možno imeti obe matriki programirni, kot to vidimo na sliki 4.5. V tem primeru imamo FPLA-postavljanje (field-programmable logic array) logičnih funkcij. F vedno pomeni visoko stopnjo programirnosti. Na sliki vidimo programiranje enakih funkcij f_1 do f_4, kot smo jih imeli na predhodnih dveh slikah.

4.2 PAL-LOGOČNE SHEME

4.2.1 Programirna matrika

Logične sheme, ki se nanašajo na PAL-čipe, se riše tako, da se da poudarek programirnemu delu sheme. Izhodi dekodirne matrike so konjunktivne zbi-



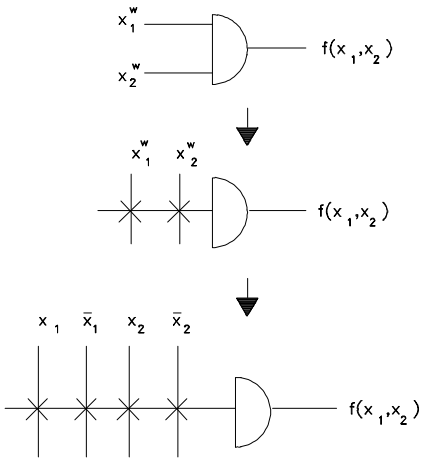
Slika 4.5: FPAL-princip programiranja logičnih funkcij.

ralnice, na katerih disjunktivno odvzemamo konjunkcije tako, da pridemo do DNO obravnavane logične funkcije. Na sliki 4.6, zgoraj, vidimo navadna konjunktivna vrata, ki dajo na svojem izhodu katerokoli funkcijo dveh spremenljivk $f(x_1, x_2)$ pri tem, da sta na vhodu spremenljivki x_1^w in x_2^w. Pri $w = 1$ imamo opraviti s spremenljivko x in pri $w = 0$ s spremenljivko $\bar{x}$. V sredini slike imamo programirni točki pri spremenljivkah x_1^w in x_2^w, ki dasta na izhodu tudi vse možne funkcije dveh spremenljivk. V spodnjem delu slike vidimo programirno točko za vsako spremenljivko posebej. Konjunktivni simbol na koncu vodoravnice pomeni le to, da je opazovana vodoravnica konjunktivna.

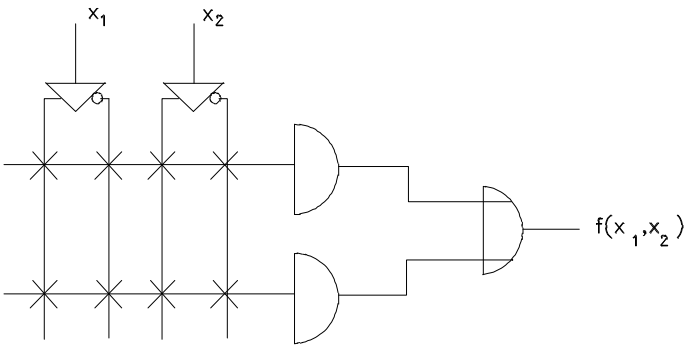
Kako nizamo skupaj konjunktivne linije v konjunktivno polje (zbiralnice, dekodirno matriko itd.) vidimo na sliki 4.7. Če pustimo matriko na tej sliki nespremenjeno, imamo tako na prvi kot drugi vodoravnici Booleovo konstanto $x_1\bar{x}_1x_2\bar{x}_2 = 0$.

4.2.2 Upoštevanje programirnih spremenljivk

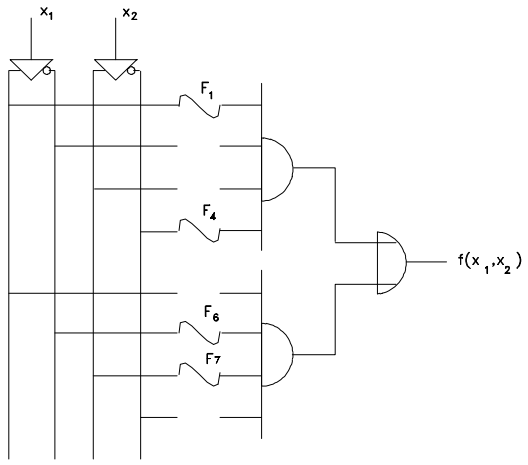
V logičnih funkcijah, ki jih realiziramo s PAL-gradniki nastopajo neodvisne spremenljivke $x_i, i = 1, 2, ..., n$. Vendar je vsaka programirna točka (F, "fuse") tudi logična spremenljivka, ki je neodvisna od x_i v toliko, kolikor je načrtovalcu dana možnost, da ji spremeni stanje. Na sliki 4.8 imamo



Slika 4.6: Prikaz konjuktivne horizontale v logični shemi.



Slika 4.7: Sestavljanje konjunktivnih linij v matriko.



Slika 4.8: Upoštevanje programirnih točk v logični funkciji.

dve neodvisni spremenljivki x_1 in x_2 ter osem programirnih točk. Točke F_1, F_4, F_6 in F_7 so nespremenjene, medtem ko je ostalim stanje spremenjeno. Kakšna je logična funkcija, ki upošteva tako x kot tudi F spremenljivke, $f(x_1, x_2, F_1, F_2, ..., F_8)$?

$$\begin{aligned} f(x_1, x_2, F_1, F_2, ..., F_8) &= (x_1 \vee \bar{F}_1)(\bar{x}_1 \vee \bar{F}_2)(x_2 \vee \bar{F}_3)(\bar{x}_2 \vee \bar{F}_4) \vee \\ &\vee (x_1 \vee \bar{F}_5)(\bar{x}_1 \vee \bar{F}_6)(x_2 \vee \bar{F}_7)(\bar{x}_2 \vee \bar{F}_8). \end{aligned} \tag{4.2}$$

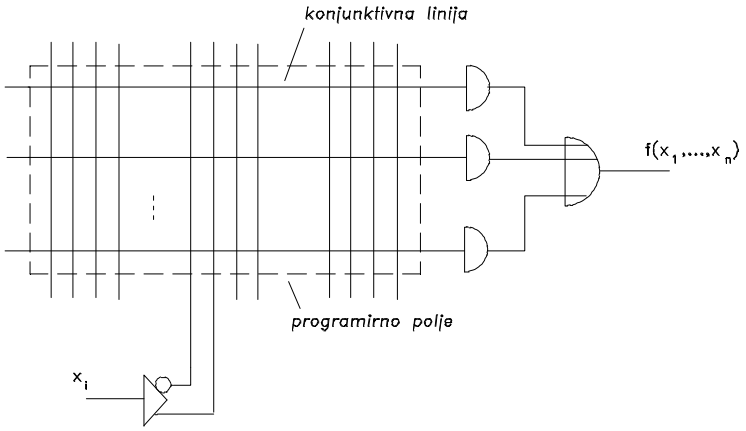
Ta funkcija je univerzalna funkcija. Lahko realizira katerokoli funkcijo dveh spremenljivk x_1 in x_2. Pri izbiri programiranja na sliki 4.8 imamo

$$F = \begin{bmatrix} F_1 & F_2 & F_3 & F_4 \\ F_5 & F_6 & F_7 & F_8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Z vstavljanjem F v (4.2), dobimo

$$f(x_1, x_2) = x_1 1\, 1\, \bar{x}_2 \vee 1\, \bar{x}_1\, x_2\, 1 = x_1 \bar{x}_2 \vee \bar{x}_1 x_2.$$

Na določen način Booleova matrika F že predstavlja osrednji del programa za postavitev logične funkcije.



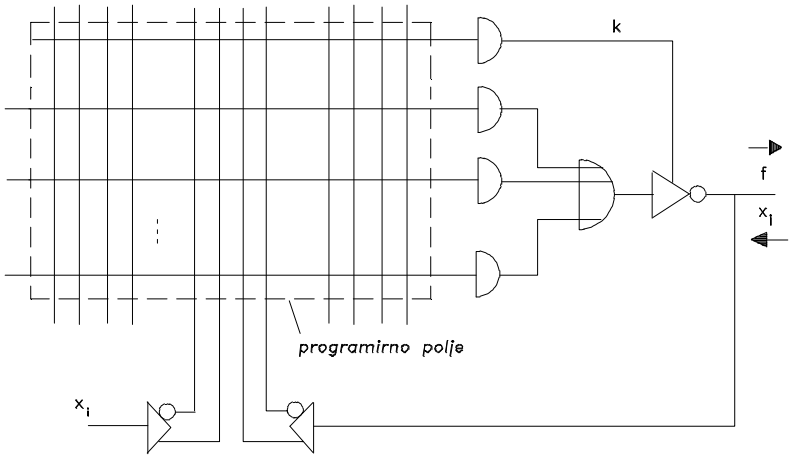
Slika 4.9: Izhodna PAL-celica.

4.2.3 Celice v PAL-gradnikih

Normalne razsežnosti PAL-struktur zahtevajo čipe z 20 priključki. Največ teh priključkov odpade na vhodne spremenljivke (po številu 10, ..., 16), ostali pa so namenjeni izhodnim funkcijam (po številu 1, ..., 8), kontroli in napajanju. Čeprav lahko s programiranjem PAL-čipov dosežemo kakršnokoli logično funkcijo, imajo posamezni čipi določeno nagnjenost do določenega tipa funkcije. Zadnje dosežemo s celico z večjo programirno sposobnostjo, ki je vgrajena v čip. Poglejmo si značilnosti takšnih celic.

Najbolj enostavno uporabo PAL-strukture imamo tedaj, ko so vse vhodne spremenljivke neodvisne in signali lahko samo napredujejo. Na sliki 4.9 vidimo konjunktivno programirno polje, pri katerem se vsaka konjuktivna linija konča z vhodom na izhodni disjunkciji. Takšna celica omogoča poljubno logično funkcijo v DNO, ki je v domeni programirnega polja 32×8. Če smo bolj natančni bi morali reči, da se programirno polje dodaja k celici tako, da atribut celice prej glasi na značilnosti izhoda kot programiranje. Polje je enosmerno, brez povratnih informacij od izhoda nazaj na vhode. Vsaka vhodna spremenljivka zaseda dve vertikali, če je le potrebna v originalni in negirani obliki. V omenjeni okvir celice spada tudi celica reda $(2 \times 8) \times 32$ v PAL-čipu PAL16L2. Omenjene vhodne dimenzije dajo vedeti, da je na PAL-čipih tudi veliko vezij za prireditev spremenljivke x^w.

Načrtovalci PAL-čipov so zgodaj spoznali, da lahko prihranijo precej

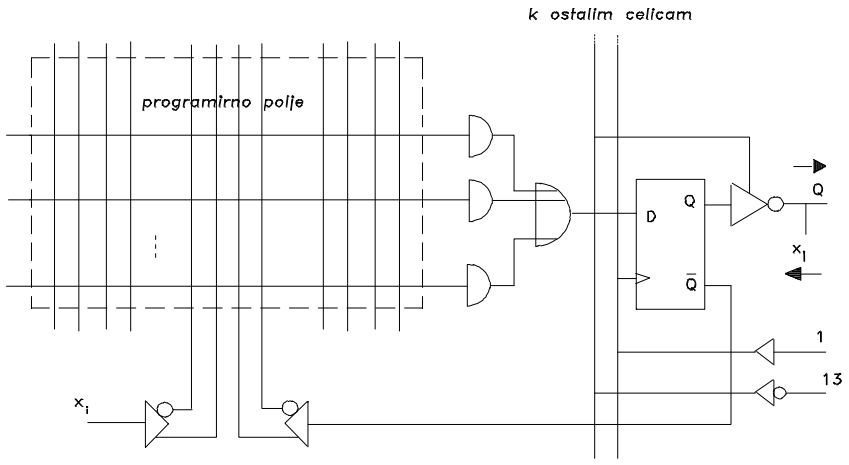


Slika 4.10: PAL-celica z vhodno-izhodnimi priključki.

priključkov, če izhode spremenijo v vhode-izhode (3-stanjsko vezje). Tako celico, ki ima vhodno-izhodne priključke vidimo na sliki 4.10. V samem programirnem polju vzamemo konjunktivno linijo s spremenljivko k, ki jo vodimo v 3-stanjsko vezje. Če je $k = 1$, ima ustrezni priključek vlogo izhoda in če je $k = 0$ pa vlogo vhoda. Spremenljivko x_j uporabljamo prav tako kot spremenljivko x_i (glej sliko 4.10). Celico z omenjeno značilnostjo imenujemo vhodna-izhodna celica. Z razvojem programirnih čipov (na splošno gledano) je na programirnih čipih vedno več vhodno-izhodnih priključkov.

Naslednja celica je registrska celica. To dobimo iz celice na sliki 4.10 tako, da pred 3-stanjsko vezje vnesemo D-pomnilno celico. Izhod celice vodimo kot izhod čipa (Q), hkrati pa ga vodimo tudi, kot vhod, nazaj na programirno polje ($\bar{Q}$). Takšno celico oziroma programirno možnost vidimo na sliki 4.11. D-celice vseh PAL celic sinhroniziramo z uro na priključku 1, medtem ko izhode Q vseh PAL-celic naenkrat, kot register, vežemo na izhode čipa s signalom na priključku 11. Na primer čip PAL16R8 vsebuje 8 takšnih registerskih celic, kar zadostuje za en podatkovni zlog.

Če delamo na področju računalniške aritmetike, je dobro, da imamo vsaj na nivoju MSI gradnike za načrtovanje takšnih enot. PAL-tehnika ima tak gradnik, za katerega je značilna aritmetska PAL celica, ki jo vidimo na sliki 4.12. Ta celica realizira vse funkcije dveh spremenljivk A in B, vključno s konstantama 0 in 1.



Slika 4.11: Registrska PAL-celica.

Zgled 5 *:*

Kako pridemo v prvi vrstici na sliki 4.12 *do Booleove konstante* 1*, če upoštevamo tudi programirne spremenljivke F?*

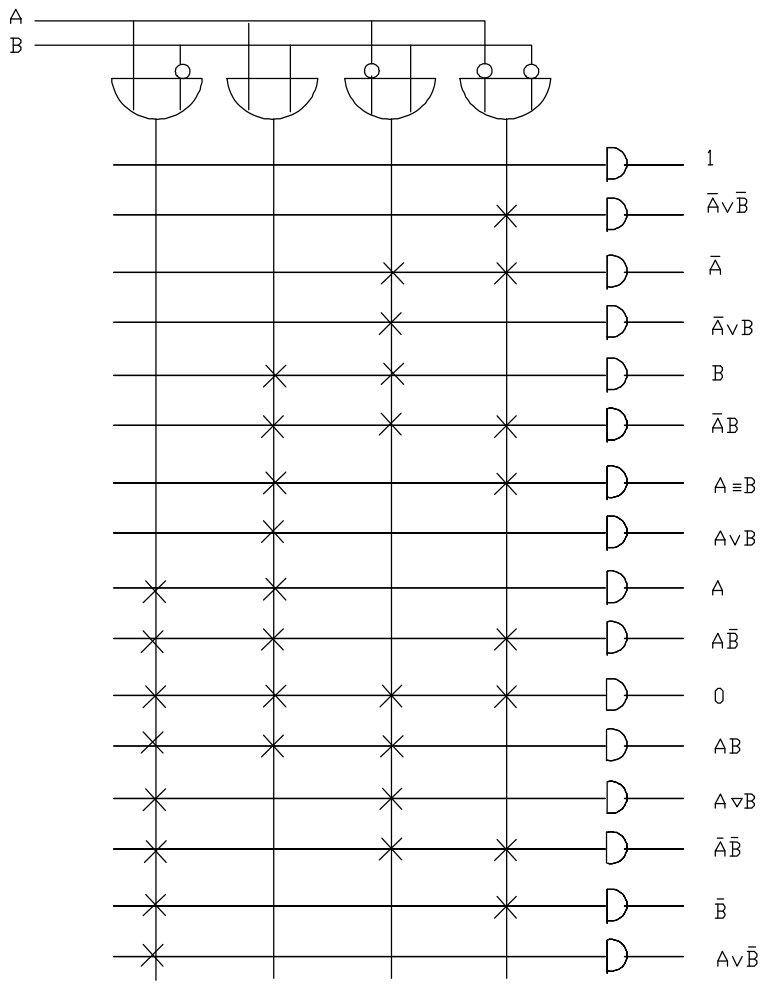
Na osnovi izraza (4.2) *in slike* 4.8 *lahko zaključimo, da imamo štiri programirne upore (za vsako vertikalo imamo enega) z vrednostmi* $F_1F_2F_3F_4 = 0000$*, saj imajo vsi štiri programirne točke spremenjeno stanje. Od tod sledi, da imamo*

$$\begin{aligned} f(A,B) &= (A \vee \bar{B} \vee \bar{F}_1)(A \vee B \vee \bar{F}_2)(\bar{A} \vee B \vee \bar{F}_3)(\bar{A} \vee \bar{B} \vee \bar{F}_4) \\ &= (A \vee \bar{B} \vee 1)(A \vee B \vee 1)(\bar{A} \vee B \vee 1)(\bar{A} \vee \bar{B} \vee 1) \\ &= 1\,1\,1\,1 = 1. \end{aligned}$$

4.3 NAČRTOVANJE S PAL-TEHNIKO

4.3.1 Uvod

Če hočemo načrtovati RSS s PAL-čipi, moramo imeti pred seboj katalog vseh PAL-čipov. Teh je okrog 25 (MM, PAL serija 20 in 24). Programiranja se lahko lotimo ročno ali računalniško, z orodji, kot so ABEL, PALASM,



Slika 4.12: Aritmetična PAL-celica.

TOABEL (konverzija PALASM v ABEL). Naloga načrtovalca je, da zna za podano logično funkcijo izbrati primeren PAL-čip ali čipovje. Pri tem moramo zlasti paziti na tipičnost PAL-funkcij, ker takšni čipi bolj optimalno pokrijejo funkcijo, če se le ta dovolj ujema z omenjeno tipičnostjo. Ta izhaja že iz notacije oziroma imena čipa.

Ime čipa je sestavljeno iz naslednjih simbolih

PAL N X M C J 883Y

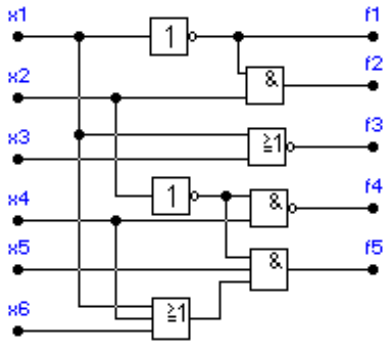
PAL: družina programirnih čipov
N: število vhodov logične površine
10, 12, 14, 16, 18, 20
X: vrsta logike:
H: H-logika
L: L-logika
C: komplementarna logika
X: XOR-čip z registrom
A: aritmetski čip z registrom
R: registerski čip
M: število izhodov čipa
1 do 8
C: temperaturno področje
C=0 do 75 ^{0}C
M=$-$55 do 125 ^{0}C
J: ohišje
J: keramično DIP-ohišje
N: plastično DIP-ohišje
F: sploščeno (flat-) ohišje
Y: US standard MIL-STD-833 (metodi Y: B, C).

Oglejmo si tako ročno kot tudi avtomatično programiranje PAL-čipov.

4.3.2 Ročno programiranje

Dana je logična shema na sliki 4.13. Ta je v neintegrirani obliki in predstavlja izsek iz večje logične sheme, ki naj bo realiziran z enim PAL-čipom.

PAL-čipi se izdelujejo tako v H- kot tudi v L-logiki. Sistem izhodnih funkcij se glede na H in L razlikuje, kot prikazujete naslednja zapisa:

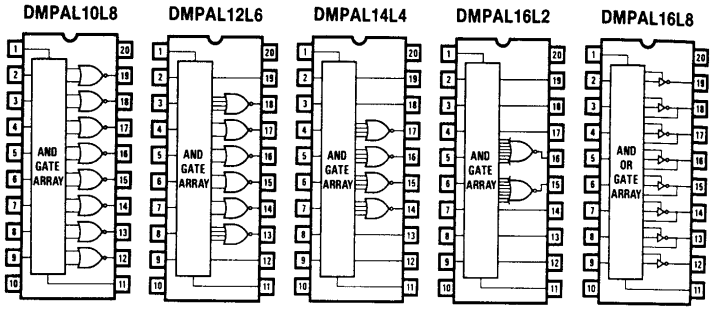


Slika 4.13: Slučajna logična shema, ki jo želimo položiti v PAL-čip.

$$
\begin{aligned}
&\text{H-}\textit{logika}\text{:}\\
&f_1 = \bar{x}_1\\
&f_2 = \bar{x}_1 x_2\\
&f_3 = x_1 \vee x_3\\
&f_4 = \overline{\bar{x}_2 x_4}\\
&f_5 = \bar{x}_2 x_5 (x_1 \vee x_4 \vee x_6)
\end{aligned}
\tag{4.3}
$$

$$
\begin{aligned}
&\text{L-}\textit{logika}\text{:}\\
&\bar{f}_1 = x_1\\
&\bar{f}_2 = x_1 \vee \bar{x}_2\\
&\bar{f}_3 = \bar{x}_1 \bar{x}_3\\
&\bar{f}_4 = \bar{x}_2 x_4\\
&\bar{f}_5 = x_2 \vee \bar{x}_5 \vee (\bar{x}_1 \bar{x}_4 \bar{x}_6)\,.
\end{aligned}
\tag{4.4}
$$

Odločimo se za L-logiko, zato moramo vse nadaljnje delo vezati na sistem (4.4). Za ta sistem enačb imamo $N = 6$ in $M = 5$. Tako nam bi zadostoval PAL-čip PAL6L5, vendar takšnega ne proizvajajo. Vzamemo takšnega, ki ga proizvajajo in ki v največji meri pokriva sistem (4.4). Tak je na primer PAL10L8. Imamo torej redundanco na vhodu in izhodu, kar pa nas preveč ne moti. Na sliki 4.14 vidimo okolje čipa PAL10L8. Iz prvega primerka na sliki izhaja, da tega čipa ne moremo uporabiti, ker ima na svojem izhodu



Slika 4.14: Pet primerkov PAL-čipov.

samo 2-vhodna (Pierceova) vrata, v sistemu (4.4) pa potrebujemo tudi 3-vhodna (funkcija f_5). Naslednji čip na sliki 4.14 je PAL12L6. Ta ima dvoje 3-vhodna vrata na svojih izhodih, zato ga lahko izberemo za realizacijo našega sistema enačb.

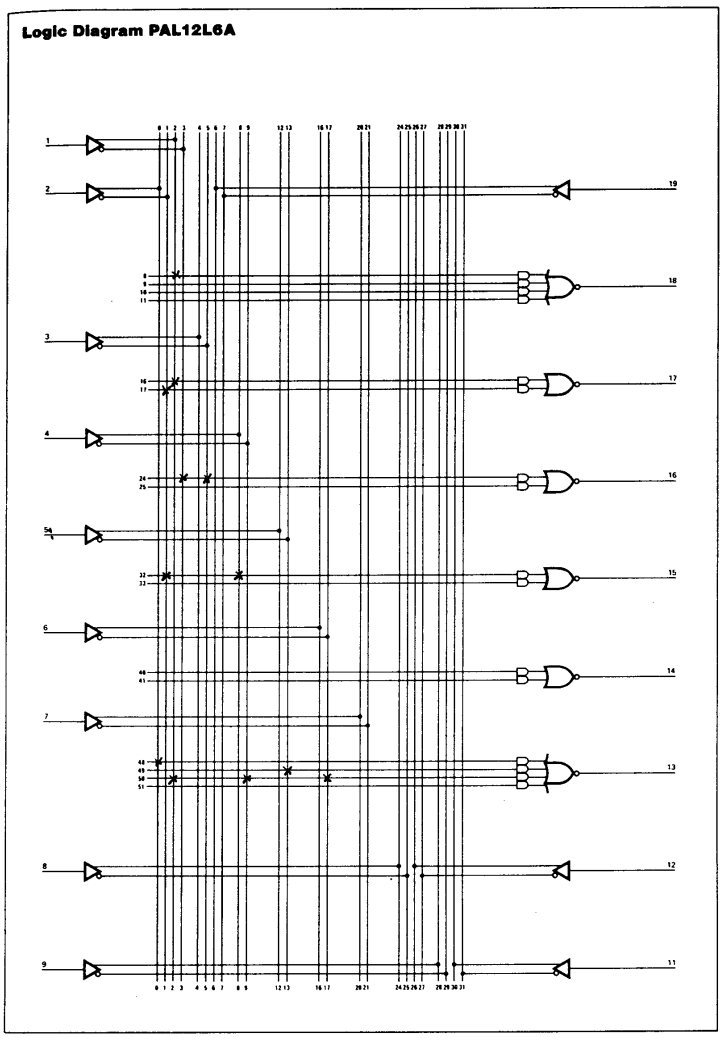
Zapišimo sistem enačb (4.4) v bolj primerni obliki za PAL-čipe. Tako imamo

$$\begin{aligned} \bar{O}_1 &= I_1 \\ \bar{O}_2 &= I_1 \vee \bar{I}_2 = \overline{\bar{I}_1 I_2} \\ \bar{O}_3 &= \bar{I}_1 \bar{I}_3 \\ \bar{O}_4 &= \bar{I}_2 I_4 \\ \bar{O}_5 &= I_2 \vee \bar{I}_5 \vee \bar{I}_1 \bar{I}_4 \bar{I}_6 \,. \end{aligned} \tag{4.5}$$

Vhodne in izhodne priključke i na čipu izberemo za naš primer takole:

$$\begin{array}{ll} 1 : I_1 & 18 : O_1 \\ 2 : I_2 & 17 : O_2 \\ 3 : I_3 & 16 : O_3 \\ 4 : I_4 & 15 : O_4 \\ 5 : I_5 & 13 : O_5 \\ 6 : I_6 & \end{array} \tag{4.6}$$

Na sliki 4.15 nam znaki $\times$ pomenijo intaktnost programirnih točk, zaprogramirane pa so točke (presečišča vodoravnic z navpičnicami) brez posebnega znaka. Dobljena slika je zadosten napotek za postavitev vhodnih po-



Slika 4.15: Programiranje čipa PAL12L6 za sistem enačb (4.6).

datkov programatorja, na katerem je, kot izhod, čip PAL12L6. Vse točke, ki za naš sistem logičnih funkcij ohranjajo nespremenjeno stanje, so:

(8,2), (16,2), (17,1), (24,3), (24,5), (32,1),
(32,8), (48,0), (49,13), (50,2), (50,9), (50,17),

vsem ostalim so spremenjena stanja.

Normalna iztočnica za programiranje je programirni list, na katerem so vpisani vsi programirni upori, tako tisti, ki jih je zaprogramiral proizvajalec in tisti, ki so na voljo kasnejšemu načrtovalcu, da z njimi zaključi funkcionalnost čipa. Tak prazen programirni list za čip PAL12L6 vidimo na sliki 4.16. Ker je v določeni logični shemi lahko veliko PAL12L6-čipov, imamo ob strani programirnega lista statistiko, ki je potrebna pri postavljanju razvoja in proizvodnje logične sheme s PAL-čipi.

Na programirnem listu za opazovani čip je v celoti $32 \times 64 = 2048$ bitov logične površine, od katerih je lahko vsak v stanju L ali H ali nezapisan bit. Vse L in H postavke postavlja proizvajalec čipa tekom proizvodnje. Na prazna mesta načrtovalec RSS lahko vpisuje $\mathcal{L}$. Če stori, ostane programirna točka intaktna, medtem ko prazno mesto pomeni, da je točka postavljena na $\mathcal{H}$. V primeru $\mathcal{H}$ se programski upori postavijo na veliko upornost (povezave se prekinejo). Z lastnim programiranjem lahko dosežemo le 16 horizontal in 23 vertikal in to nepopolnih. Vse konfiguracije, ki jih omogoča opazovani čip, dobimo torej z manj kot 290 programirnimi točkami, čeprav čip v celoti obsega 2048 takšnih točk. Tovarniško postavitev H in L je cenejša od $\mathcal{L}$-postavitve, ki jo napravi kasnejši načrtovalec.

Na sliki 4.17 so postavljena stanja $\mathcal{L}$ za sistem funkcij (4.4). S tem smo zaključili postavitev PAL-čipa, sledi le akcija na samem programatorju, ki zapiše želene podatke v čip. Za programiranjem opazovanega čipa gremo k naslednji shemi, ki ustreza sliki 4.13 za ravnokar opazovani čip. Itd.

4.3.3 PAL-programiranje z orodjem ABEL

Za PAL-tehniko načrtovanja sta najprimernejši računalniški orodji ABEL in starejši PALASM. V tem razdelku si poglejmo le glavne faze načrtovanja v okolju ABEL. Ta nam omogoča:

- stalno sintaksno kontrolo logičnih shem,
- verifikacijo, da se da zasnovano shemo implementirati,
- minimizacijo (redukcijo) logičnih shem,
- simulacijo logičnih shem,

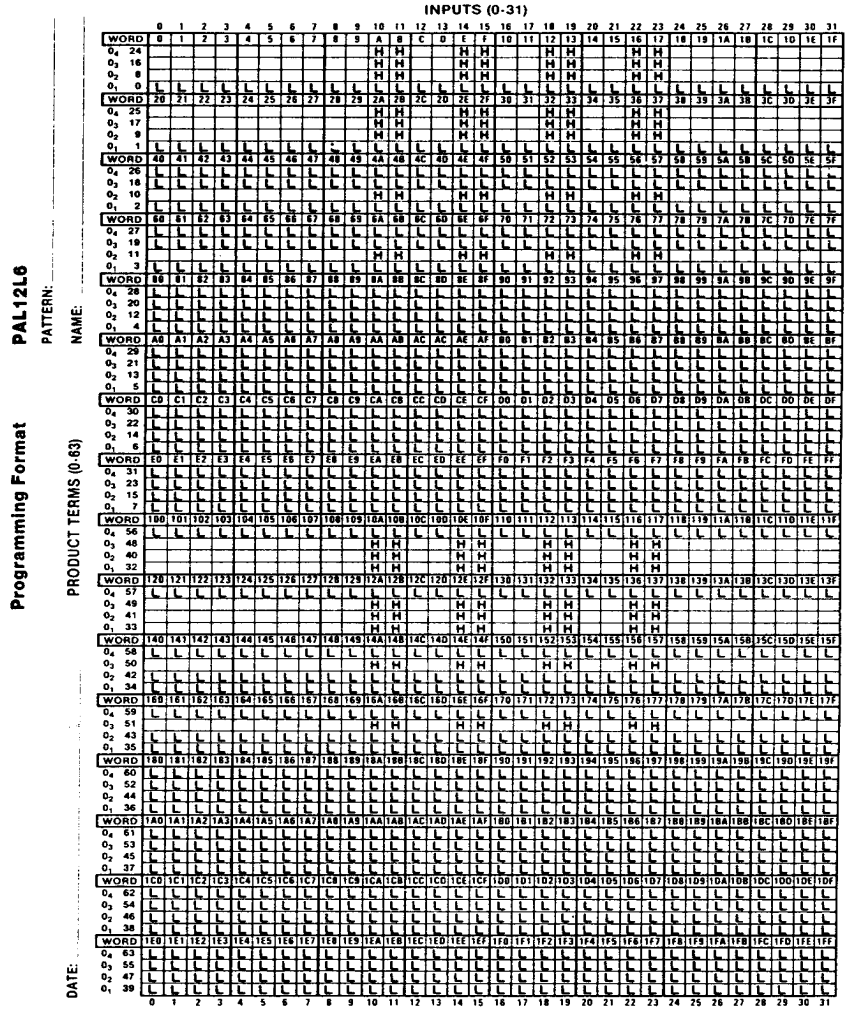


Figure 20. Blank programming Format Sheet for the 12L6 PAL.®

Slika 4.16: Programirni list čipa PAL12L6.

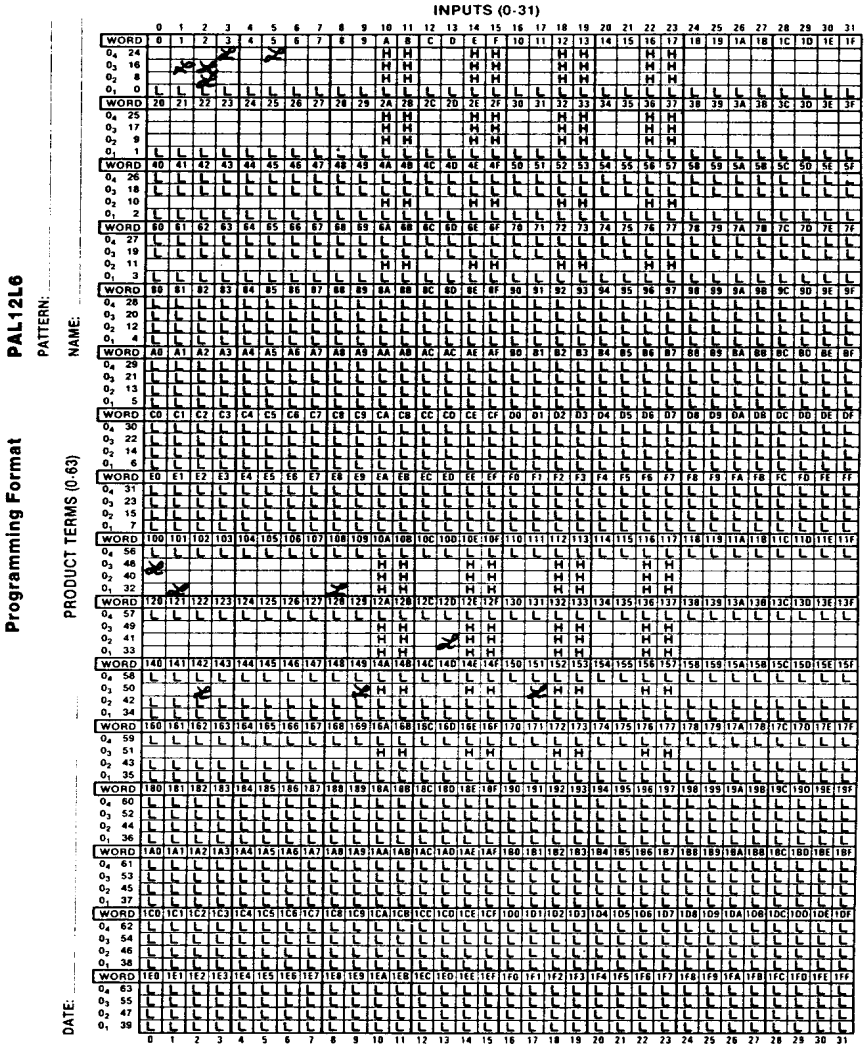


Figure 20. Blank programming Format Sheet for the 12L6 PAL.®

Slika 4.17: Izpoljnjeni programirni list za čip PAL12L6.

- kreiranje datoteke v izhodnem JEDEC,
- dokumentacijo snovanja,
- funkcionalne teste itd.

Po podpori ABEL posežemo takrat, ko imamo opraviti s programirno logiko (PAL, PLD, FPGA, PROM), torej takrat, ko je na izhodu delovne postaje programator, ki postavi dokončno čip kot realizacijo obravnavane logične sheme. Bistvene faze načrtovanja so:

∘ priprava (PARCE)
∘ prevajanje (TRANSOR)
∘ optimizacija (REDUCE)
∘ logične simulacije (SIMULATE).

Po teh fazah nam je znano ali je z logičnega stališča s čipom vse v redu. Ko je takšno delo za vse čipe opazovane sheme končano, opravimo še fazi:

∘ programirna slika (FUSEMAP) in
∘ dokumentacije (DOCUMENT).

Seveda dokumentacije ne delamo toliko časa, dokler simulacija in testiranje ne dasta zadovoljivih rezultatov. Čipe programiramo, ko je dosežena vsa dokumentacija brez vsake napake.

V skladu z gornjimi fazami imamo na voljo naslednje tipe datotek:

– `XXX.ABL`
– `XXX.DOC`
– `XXX.SIM`
– `XXX.LST`
– `XXX.REP` in
– `UNN.JED`.

`XXX.ABL` je izvorna datoteka, ki jo lahko pripravimo z navadnim urejevalnikom besedil (WS, WORD). Izvajalni program `ABEL.COM` izvaja izvorno datoteko in na tak način pridemo do vseh ostalih zgoraj omenjenih datotek. Izhodna datoteka za programiranje čipa je JEDEC-datoteka, s končnico `JED`. Pri tem nam ime `UNN` lahko pomeni U*nn*, kar je naslov enote (na primer PAL-čip) v večji logični shemi. Končnica `SIM` pomeni, da gre za datoteko s simulacijskimi podatki, REP je končnica reporta, `LST` predstavlja dokumentacijsko listo in `OUT` pomeni, da gre za datoteko s testnimi podatki.

Poglejmo si, kakšno izvorno datoteko XXX.ABL potrebuje ABEL? Če hočemo priti do nje, se moramo dosledno držati sintaksnih navodil za en modul oziroma posamezne module. V primeru enega modula (enega programirnega elementa) imamo

```
MODULE modname (dummy_arg dummy_arg ...  )
```

```
FLAG statement ...
TITLE statement
declarations
EQUATIONS | TRUTH_TABLE | STATE_DIAGRAM |...
TEST_VECTORS ...
END modname;
```

V primeru več modulov v eni logični shemi si v izvornem programu moduli enostavno sledijo. Tedaj velja sintaksa

```
1st Module Start
Flags
Title
Declarations
Constant Declarations
Macro Definitions
Device Declarations
Pin and Node Assignments
Attribute Declarations
Library References
Boolean Equations | Truth Tables |State Diagrams
Test Vectors
1st Module End
2nd Module Start
....
....
2nd Module End
....
```

Iz zadnjih programskih zapisov izhaja, da ABEL obvladuje makroje, referenčno knjižnico, logične enačbe, pravilnostne tabele, diagrame prehajanja stanj (za tiste gradnike, ki vsebujejo pomnilne celice) in diagnostiko v smislu testnih vektorjev. Za vsako navedeno postavko imamo ustrezno sintakso in navodila.

Vzemimo logično shemo na sliki 4.13 in poskušajmo najti njeno PAL-realizacijo, podobno kot smo to napravili pri ročnem načrtovanju. Izvorna datoteka je naslednja

```
MODULE primer
TITLE 'GRUPA VRAT'
      U23 device 'PAL12L6'
```

```
   I1,I2,I3,I4,I5,I6 pin 1,2,3,4,5,6;
   O1,O2,O3,O4,O5    pin 18,17,16,15,13;
   H,L,X = 1,0,.X.;
   VHOD = [I1,I2,I3,I4,I5,I6];
EQUATIONS
   !O1 = I1;
   !O2 = I1#!I2;
   !O3 = !I1&!I3;
   !O4 = !I2&I4;
   !O5 = I2#!I5#!I1#!I4&!I6;
TEST VECTOR (VHOD -> [O1,O2,O3,O4,O5])
   bb000000 -> [H,L,L,H,L];
   bb111111 -> [L,L,H,H,L];
end primer
```

Simbol ! pomeni negacijo, # disjunkcijo in & konjunkcijo. Poleg teh logičnih operacij ABEL pozna še XOR ($) in ekvivalenco (!$). Zanimivo je, da dela z DNO in KNO logičnih funkcij, čeprav so logični operatorji na čipih zelo velikokrat Shefferjevi in Pierceovi operatorji.

Po izvajanju gornjega programa dobimo med drugimi datotekami tudi najpomembnejšo (podatki za programator čipa) PRIMER.JED

```
   ABEL 6.10 Data I/O Corp.
   JEDEC file for:  P12L6 V9.0
   Created on:  Mon Mar 30 11:55:24 1998
   skupek vrat
   *
   QP20* QF384* QV2* F0*
   X0*
   NOTE Table of pin names and numbers*
   NOTE PINS I1:1 I2:2 I3:3 I4:4 I5:5 I6:6 O1:18 O2:17
   3:16 O4:15 O5:13*
   L0000 11011111111111111111111*
   L0096 11011111111111111111111*
   L0120 10111111111111111111111*
   L0144 11101011111111111111111*
   L0192 10111111011111111111111*
   L0288 01111111111111111111111*
   L0312 11111111110111111111111*
   L0336 11101111101110111111111*
```

```
V0001 000000XXXNXXLXHLLHXN*
V0002 111111XXXNXXLXHHLLXN*
C1780*
7DAB
```

Zvezdica * v gornji datoteki pomeni konec določenega polja informacij tako, da moramo gledati predvsem, kakšne podatke imamo med zvezdicami. `QP20` pomeni, da gre za čip s 20 priključki, `QF384` pomeni, da je vseh dostopnih programirnih točk 384. Te tvorijo matriko 16×24 (16 konjuktivnih vodoravnic in 24 vhodnih navpičnic). Za podatki programiranja sledita vrstici z vektorjema `V0001` in `V0002`, ki predstavljata testno polje. V testnih vrsticah so testne izhodne vrednosti `H` - test na H, `L` - test na L in `N` - netestiran izhod. Za zadnjim testnim poljem je še vsota parnosti kot kontrola. Ta kontrola je nujna, saj je dovolj, da je v programirni matriki 16×24 samo en bit napačen, pa je čip PAL12L6 neuporaben.

ABEL se nahaja na tržišču že svojih 14, 15 let. Vsaka naslednja verzija zna nekaj več. V času uporabnosti verzije V.6, ABEL predstavlja HDL, ki je primeren za PLD firm Lattice, Cypress, Atmel, TI, Altera, Philips, AT&T (komponente flash), AMD (komponente mach), Lattice (komponente pLSI) itd. Glede na to široko paleto mora znati sodelovati z najrazličnejšimi orodji, kot so XACT, MAX Plus, FLEX, 32000DX, ORCA itd. Iz tega izhaja, da je programiranje PAL-čipov postal takorekoč najmanjši del uporabe orodja ABEL, čeprav se je postavljalo v zlati dobi PAL-tehnike prav za gradnike PAL.

Poglavje 5

PPAL-GRADNIKI

5.1 UVOD

Osnovna PAL-tehnika se nanaša na bipolarno tehnologijo, ki je sicer hitra, je pa zato energetsko potratna. Zadnje je slaba popotnica, če bi hoteli s PAL-tehniko, seveda z razvojem mikroelektronike, preiti iz velikostnega reda gradnikov MSI na LSI. Naj napravimo primerjavo med gradnikom PAL20 in na primer EP 300, to je PLD, ki ga lahko tako programiramo, da logično deluje kot katerikoli PAL20. Iz tabele 5.1 izhaja, da v primeru PLD potrebujemo za isto logiko dosti manjšo energijo, da lahko PLD reprogramiramo, medtem ko tega pri PAL ne moremo itd.

Če hočemo priti do večjih energetskih prihrankov, je potrebno preiti iz bipolarne na MOS tehnologijo. Seveda mora biti zadnja dovolj hitra, da nimamo prepočasnih gradnikov, saj je velika hitrost izrazita prednost prav bipolarne tehnike. V tem smislu je prišlo po 1983 letu do MOS reprogramirne PAL-tehnike, ki se imenuje GAL (generic array logic). Uporablja tehnologijo E^2CMOS, ki ima majhno porabo, veliko hitrost delovanja, veliko tehnološko gibkost, elektronsko identifikacijo, veliko zanesljivost in, kar je tudi pomembna postavka, 100 % testno sposobnost. Glede na bipolarno tehnologijo reducira porabo energije na 50 do 75 % glede na bipolarno tehnologijo in to pri hitrosti ure $5 - 30$ ns. Orodja, ki jih lahko uporabljamo v primeru GAL-tehnike načrtovanja RSS, so: ABEL, CUPL, LOG/iC,OrCAD-PLD, PLDesigner in TANGO-PLD. Če že imamo PAL-logično shemo RSS jo lahko z omenjenimi orodji spremenimo v ekvivalentno GAL-logično shemo. Seveda je ekvivalenca le logična, tehnološko sta shemi izrazito različni.

Ker je GAL zaščiteno ime firme Lattice za programirne MOS-čipe, ki

Parameter	EP 300	PAL20
programirno polje	AND-OR	AND-OR
poraba energije	40 mA	180 mA
število vhodov	17	16
število izhodov	8	8
število vhodov pr. polja	36	32
število D-pom. celic	8	8
število 3-stanjskih izhodov	8	8
postavljanje kontrole	da	ne
brisanje kontrole	da	ne
reprogramiranje	da	ne

Tabela 5.1: Prednost PLD-gradnika pred PAL.

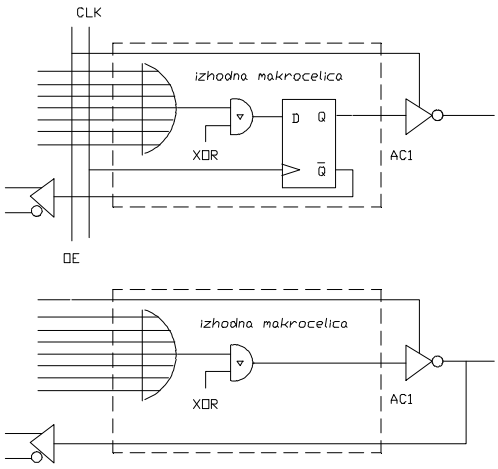
logično pokrivajo PAL-čipe, ga ne moremo uporabiti za splošno, ko mislimo hkrati na vse proizvajalce takšnih čipov. Glede na to bomo uporabili ime PPAL-gradnik, ki predstavlja omenjeni programirni (tudi brisalni in električno brisalni) PAL-gradnik ne glede na proizvajalca.

5.2 PPAL-CELICE

PPAL-čipi za gradnjo logičnih shem imajo podobno kot v PAL-tehniki programirna polja (programirni modul) in celice. Za tipični primer PPAL-gradnikov vzemimo GAL-čipe. Osrednja njhova značilnost je izhodna logična makrocelica OLMC (output logic macrocell) z veliko stopnjo funkcionalne gibkosti. GAL-tehnika pozna tri vrste delovanja OLMC: enostavno, kompleksno in registrsko delovanje celice. Iz enega delovanja gremo na drugo s pomočjo signalov SYN, AC0 in XOR. Na sliki 5.1 zgoraj vidimo registrsko konfiguracijo registrske celice. Ta način delovanja zahteva

SYN=0	globalna kontrola čipa
AC0=1	globalna kontrola čipa
XOR=0	L-logika
XOR=1	H-logika
AC1=0	definicija izhoda celice
Priključka 1, 11	urna in dosegljivostna kontrola registrskega izhoda.

Globalna kontrola pomeni, da se nanaša na vse OLMC v čipu. XOR določa

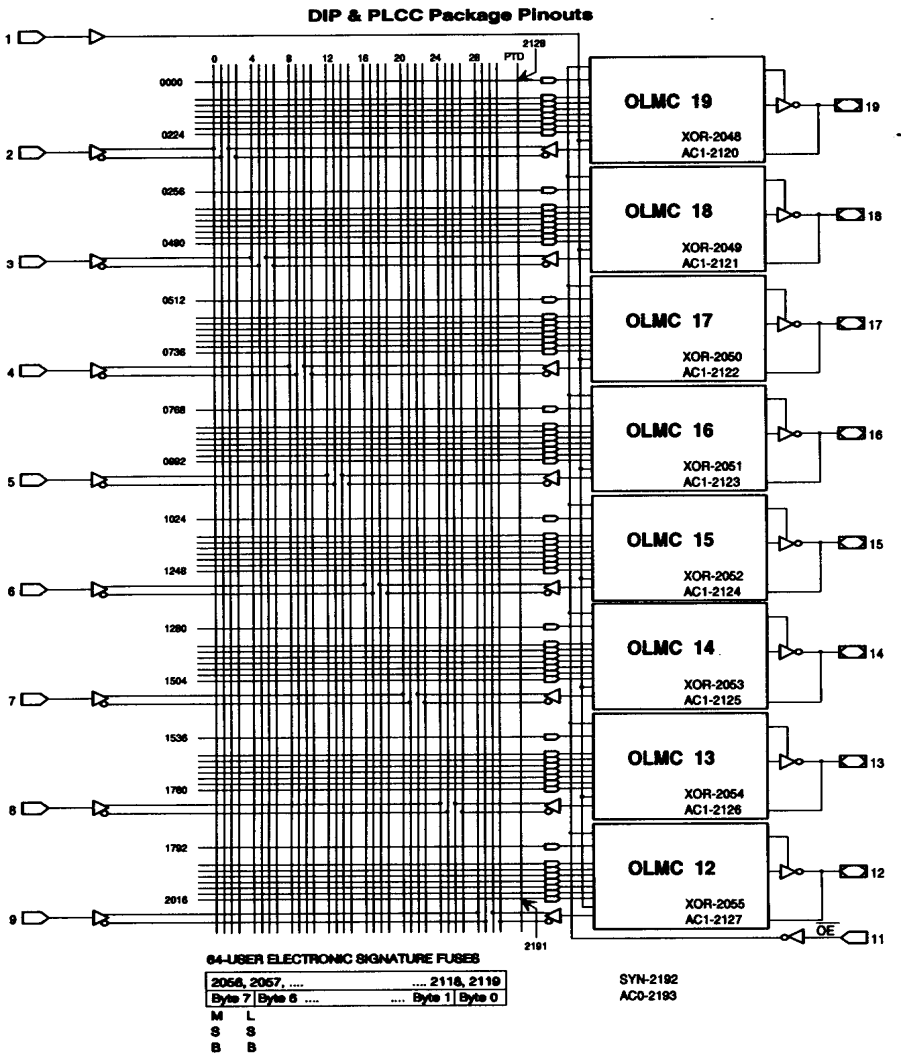


Slika 5.1: Registrski in decizijski načina delovanja registrske OLMC v GAL-čipu.

ali je aktiven spodnji ali zgornji logični nivo, medtem ko AC1 določa za vsako celico posebej njen izhod. Če hočemo preiti od izhoda zgornje sheme na izhod spodnje sheme na sliki 5.1, spremenimo samo AC1=0 v AC1=1. Ker pri spodnji shemi D-pomnilna celica nima nobene funkcije, igra registrska celica vlogo povsem decizijske celice. Pri kodi SYN AC0 = 01 imamo registrske celice, pri 11 kompleksne decizijske in pri 10 enostavne decizijske celice. Enostavna celica je značilna po tem, da je njen kontrolni vhod 3-stanjskega vezja na izhodu dan na V_{cc}, medtem ko je pri kompleksni decizijski celici ta vhod vključen v programirno polje.

Na sliki 5.2 vidimo bločno shemo čipa GAL16V8A. Programirno polje ima razsežnost $(8 \times 8) \times 32$ pri 10 vhodih in 8 izhodih-vhodih. Izhodne celice so označene OLMC i, kjer je i številka ustreznega izhodno-vhodnega priključka. V kolikor si ogledamo logično in pomnilno vsebino čipa PAL16R8 vidimo, da ni veliko razlik glede na GAL16V8A. Notacija PPAL-čipov je podobna kot pri PAL-čipih, vendar je v imenu dodatno specificirana hitrost delovanja. Tako imamo za večino PPAL-čipov približno naslednjo notacijo

PPAL N V M A - NS P C J K



Slika 5.2: Čip GAL16V8A.

PPAL: CMOS-gradnik s funkcionalnim ozadjem PAL-tehnike
GAL: ime čipov firme Lattice,
PALCE: ime čipov družina firme AMD,
PALC: ime družine čipov firme Cypress
itd.

N: število vhodov logične površine
V: tip logike čipa:
V: splošni tip
VX: XOR
RA: asinhronska-set/reset registrska logika
M: število izhodov čipa
A: tip izhoda
A: navaden izhod
B: aktivni "pull-up" na vseh priključkih
NS: hitrost delovanja
število ns
P: poraba energije
L: majhna
Q: četrtinska
C: ohišje
P: plastično DIP-ohišje
J: PLCC, "leaded chip carrier" ohišje
K: profesionalnost izvedbe
nič: navadno tržišče
I: industrijska.

Namesto NVMA določene firme navajajo številko GAL-čipa. Tako ima National Semiconductor na primer čipa GAL6001-30LNC in GAL16V8A-10, torej uporablja oba načina notacije.

5.3 ZAŠČITA PPAL-GRADNIKOV

V vrsto PPAL-gradnikov vgrajujejo tako imenovano zaščitno celico, ki preprečuje, da bi neavtorizirane osebe raziskovale in kopirale že postavljeno logično vsebino. Ko je programiranje končano, se lahko prekine dosegljivost logične vsebine čipa. Ker so na primer GAL-čipi grajeni v E^2CMOS-tehnologiji, nam seveda prekinjena dosegljivost pri naslednjem programiranju moti. Opisano zaščito lahko brišemo tekom "balk erase" procesa, ki ga vedno izvedemo pred novim programiranjem. Seveda se nam pri tem

briše tudi stara vsebina pod zaščito. Torej do stare vsebine ne moremo, lahko pa programiramo novo.

5.4 PROGRAMIRANJE PPAL-GRADNIKOV

Programiranje PPAL-čipa poteka približno tako kot smo to prikazali na sliki 4.15 za PAL-čip. Iz takšne slike gremo k zapisu izvornega programa v določenem orodju, ki nam da, seveda po procesiranju, JEDEC sliko (datoteko) za programiranje čipa na ustreznem programatorju.

Orodje, ki ga priporoča firma Lattice, proizvajalka GAL-čipov, je CUPL. To orodje vsebuje vmesnik PTOC (PALASM to CUPL Translator), kar pomeni, da pokriva tudi PAL-tehniko. Zadnje je ugodno, če PAL-rešitve nadomeščamo z manj potrošnimi GAL-rešitvami. CUPL slovi po logičnem simulatorju CSIM. Obstaja konverzija PALtoGAL, ki 100 % prevaja PAL-logične sheme v GAL-logične sheme. Konverzije v GAL-tehniko so možne tudi za EPLD in PEEL. Ne smemo pozabiti, da obstaja vrsta PAL- in podobnih tehnik, ki jih različne firme imenujejo po svoje. Tako imamo na primer AT (Atmel), PALC (Cypress), HPL (Harris), PEEL (ICT), EPL (Ricoh), CPL (Samsung), PLUS (Signetics), SPL (Sprague), TIBPAL in TICPAL (TI) in druge, ki so z orodji ekvivalentno določljive h GAL firme Lattice. Iz zadnjega lahko zaključimo, da igra Lattice dominanto vlogo na področju GAL-gradnikov.

GAL-čipi so uporabni predvsem pri: DMA-kontroli, logičnih avtomatih (state machine control), hitrem grafičnem procesiranju in seveda v standardni logiki, v kateri je bila uporabna PAL-tehnika.

Poglavje 6

ASIC-TEHNIKA

6.1 UVOD

Povedali smo že, da družina ASIC obsega popolne uporabniške gradnike (full-custom devices), standardne celice, logične razporeditve (gate arrays) in PLD (programmable logic devices), to je programirne logične gradnike. PLD so generični gradniki, podobno kot GAL-gradniki. V obeh primerih generičnost pomeni, da obstaja zaradi programirnosti s strani kasnejšega načrtovalca (uporabnika) velika možnost za uporabo določenega čipa pri najrazličnejših aplikacijah. Če je temu tako, ne smemo nekaj generičnosti odrekati tudi PAL-gradnikom.

Znana proizvajalca PLD sta Altera in Cypress Semiconductor. Oba sta uvedla EPLD na osnovi EPROM-tehnologije. Posebnost je v tem, da je v PLD-tehniki bipolarna programirna točka (takšno točko imamo pri klasičnih bipolarnih PAL-čipih) zamenjana z ustrezno MOS programirno točko. Kaj pomeni ta zamenjava? Značilnosti bipolarne točke so:

- ima visoko disipacijo energije, kar preprečuje veliko integracijo, zato ne moremo imeti veliko število programirnih točk na majhni površini,.
- ne dovoljuje 100 % testiranja (če hočemo testirati programirno točko, ji moramo spremeniti stanje, tega pa ne moremo spremeniti nazaj v prvotno stanje),
- točka se programira lahko le enkrat,
- programska orodja, ki se nanašajo na bipolarne točke so starejša in okorna.

Altera je prva firma, ki je omenjene slabosti hotela, in tudi uspela, odstraniti na osnovi MOS-tehnologije. Uvedla je CMOS "floating-gate" tehnologijo, ki ima mnogo manjšo disipacijo energije in zato dosega tudi mnogo večje čipe. Že pred dobrimi desetimi leti je bila dosežena gostota 1800 ekvivalentnih vrat na čip. Programirni princip pri EPLD izhaja iz tehnologije MOS-EPROM, ki se je uveljavljala že v sedemdesetih letih. Važna je reprogramiranost (erasability), kar pomeni, da so čipi uporabni večkrat. Po EPROM-principu brisanja imamo reprogramiranje pri UV svetlobi, ki pada na površino 1 uro z jakosti 12000 $\mu\mathrm{Wcm}^{-2}$. Poleg EPLD poznamo tudi EEPLD, pri katerem gre za električni princip reprogramiranja. Poleg PLD obstaja tudi FPGA-tehnika. Obstaja izkustveno pravilo, da o PLD-čipih začnemo govoriti, če čip obsega logično vsebino vsaj 5 MSI-čipov. Ostre meje med PLD- in FPGA-tehniko ni. PLD-tehniki se še vedno priznava določena primerljivost z bipolarno tehnologijo (na primer programirni moduli na osnovi AND-OR matrik), medtem ko so FPGA-čipi že izredno veliki s specifičnimi značilnostmi, ki z bipolarnostjo nimajo nobene primerljivosti več. V okviru FPGA govorimo o "antifuse" točki (programirni protitočki) kot dopolnilu k "fuse" točki. Obe skupaj nudita velike gibkosti pri proizvodnji in v tehniki programiranja logičnih razporeditev. Na primer FPGA-čipi firme Actel obsegajo že 750000 protitočk, pod 100000 protitočk pa logičnih razporeditev ne gradijo. Ko govorimo o teh številkah se moramo zavedati, da je programirna točka nekaj drugega kot ekvivalentna vrata. Število ekvivalentnih vrat je precej nižje od števila programirnih točk, ki se postavljajo tekom proizvodnje in kasneje s strani načrtovalca logične sheme.

V družini ASIC-gradnikov imamo čipe z gostoto 100000 ekvivalentih vrat pri dovolj veliki gibkosti ob postavljanju logičnih funkcij in logičnih shem. Firma Xilinx si je v okviru že omenjene poplave raznih tehnik izbojevala še eno tehniko LCA (logic cell array), pri kateri so pazili, da je načrtovanje logičnih shem relativno hitro, čeprav gre za večje število ekvivalentnih vrat na čip. Tudi LCA-čipi se postavljajo v tehnologiji CMOS, kar jim daje prednosti, ki jih izžareva ta tehnologija (100 % testiranje, velika gostota ekvivalentnih vrat na čip, manjša poraba energije itd.). Značilnost LCA-tehnike je v tem, da postavlja programirne celice, ki so med seboj ločene s programirnimi (povezovalnimi) vodili.

Z vidika zanesljivosti imamo za ASIC-tehniko posebej prirejen performančni indeks seveda poleg vseh ostalih zanesljivostnih indeksov, ki jih poznamo v računalništvu in elektroniki [19]. Ta indeks se imenuje 'odpoved na čas', FIT (failures in time). 1 FIT je 1 odpoved na 10^9 ur. Tako ima na

primer sistem s mikroprocesorjem s 7 FIT/PROC, 9 ASIC s 10 FIT/ASIC in 5 RAM s 15 FIT/RAM zanesljivost podano z izrazom

$$R = 7 \times 1 + 10 \times 9 + 15 \times 5 = 172 \text{ FIT}.$$

Stroškovni vidik ASIC-tehnike smo si ogledali že na koncu 1. poglavja (tabela 1.1).

6.2 NAČRTOVANJE Z ALTERA-EPLD-TEHNIKO

6.2.1 Altera-EPLD gradniki

Poznane EPLD-družine firme Altera so: EP (3XX - 18XX), MAX (5XXX -9XXX) s "cross-bar programmable interconnect array" in FLEX s "fine grain" povezovalno tehniko. Tehnološka osnova Alterinih čipov je CMOS oziroma CHMOS. Čipi s CHMOS so tisti, ki imajo končnico številke 10 sicer pa se številka nanaša na približno število ekvivalentnih vrat na čipu. CHMOS v čipih EP X10 zahteva le še 10 % porabe, ki jo imamo v logično ekvivalentni TTL-tehniki. Že pri GAL-tehniki smo videli, da čipi logično pokrivajo vse PAL-čipe, te sposobnosti pa ima tudi čip EP300 oziroma EP310 sicer pa smo ta čip primerjali s PAL čipi že v tabeli 5.1. Pri primerni postavitvi parametrov, ki so podani tudi tabelarično, lahko EP300 pokrije 16 različnih PAL-čipov.

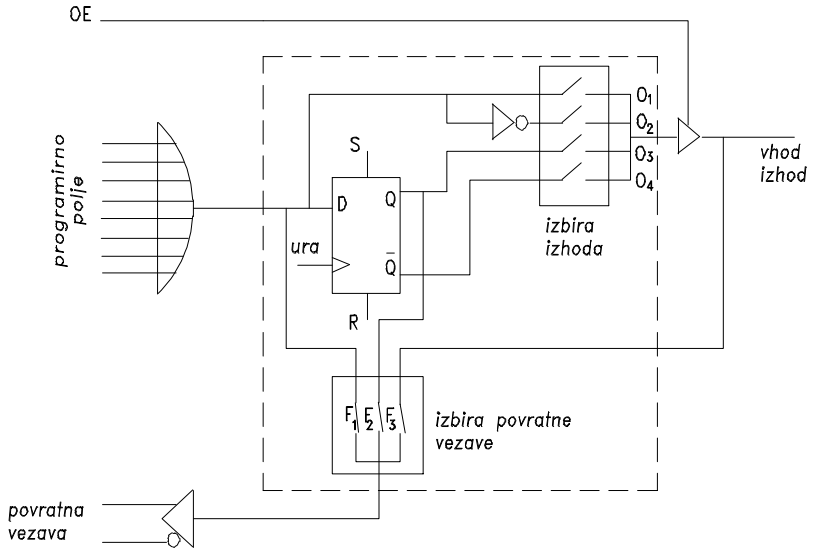
Arhitektura EP-čipov predvideva programirno polje, ki ga zaključuje funkcijsko gibko vhodno-izhodno vezje. Sestavo obeh imenujemo makrocelica. EP z večjo številko ima večjo število takšnih celic, večji čipi pa vsebujejo vodila za povezovanje večjih sklopov. Kakšno število makrocelic in drugih postavk imajo EP-čipi, vidimo v tabeli 6.1.

Selekcijsko vezje v makrocelici vidimo na sliki 6.1. Z izbiro odprtosti in zaprtosti preklopnikov O_i, $i = 1, 2, 3, 4$ izberemo izhod. Ta je lahko izhod iz programirnega polja (decizijski način delovanja makrocelice) v negirani ali nenegirani obliki ali pa izhod D-pomnilne celice (registrski način delovanja makrocelice) v negirani ali nenegirani obliki. Podobno imamo za povratek nazaj v programirno polje. Na osnovi F_1, F_2, F_3 pridemo do povratka iz programirnega polja, iz pomnilne celice ali pa iz izhodno-vhodnega priključka.

Kako so makrocelice sestavljene na čipu želimo pokazati na sliki 6.2 [13], na kateri vidimo bločno sliko srednje velikega čipa EP900. Ta vsebuje

Postavka	EP300	EP310	EP600	EP900	EP1200	EP1210	EP1800
Priključki	20	20	24	40	40	40	68
Vhodi	18	18	20	38	36	36	64
Izhodi	8	8	16	24	24	24	48
Makrocelice	8	8	16	24	28	28	48
Konjunktivne linije	74	74	160	240	236	236	480
Ekvivalentna vrata	352	352	716	1096	1176	1176	2140
Tok neaktivnosti, mA	25	20	0.01	0.01	3	3	0.01
Tok aktivnosti, mA	25	20	15	30	35	25	60
Max. hitrost	18.2	40	40	33.3	12	20	25

Tabela 6.1: Parametri Altera EP-čipov.



Slika 6.1: Selekcijsko vezje na koncu programirnega polja makrocelice.

24 makrocelic, kar pomeni, da je čip 3 krat večji od čipa EP300. Primarni vhodi so vezani na vhodna vezje, ki se na sliki 6.2 nahajajo na vrhu in spodaj. Vhodi pa so lahko tudi na izhodno-vhodnih priključkih posameznih makrocelic (leva in desna stran na sliki), če so seveda ustrezno postavljene s selekcijskimi vezji. Primer nazorno prikazuje, kako dosegajo veliko racionalnost priključkov na PLD-čipih. Leva stran makrocelic je vezana z desno stranjo preko 72 povezav (na vrhu in spodaj slike). Že čip EP900 nakazuje, na kompleksnost povezovanj makrocelic, zato še večji EP-čipi uvajajo ustrezna vodila med celicami.

6.2.2 Implementacija Altera EP-čipov v RSS

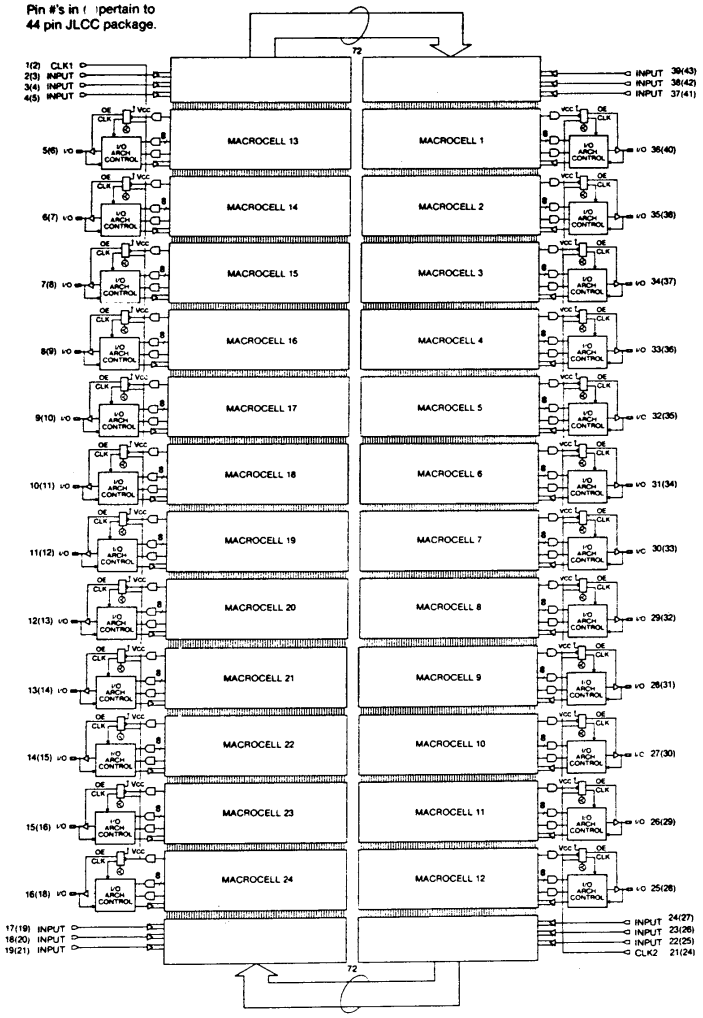
Vzemimo, da imamo 5 SIO-perifernih pristopov k CPU (na primer mikroprocesor Intel 8088) s pomnilnikom 8K RAM (6164) in 8K EPROM (2764). Potrebno je uskladiti delovanje omenjenih enot tako, da so I/O-enote 1, 2, 3 in 4 počasne in 0 hitra enota. Za kontrolno enoto je dovolj, da vzamemo EP310, ..10 zato, da imamo manj porabno in praktično enkrat hitrejše delovanje. Bločno shemo rešitve problema vidimo na sliki 6.3. Osnovna naloga PE310 je, da dekodira naslove in jih spremeni v selekcijske signale za čipe SIO0, SIO1, ..., SIO4, 6164 in 2764.

Da lahko izpišemo ustrezne logične funkcije, nam je potrebna razdelitev celotnega naslovnega prostora, ki ga omogoča 16-bitna beseda naslavljanja, porojena s strani mikroprocesorja. Ustrezno pomnilno shemo vidimo na sliki 6.4. Iz tega "memory map" prikaza izhajajo naslednje funkcije selekcije

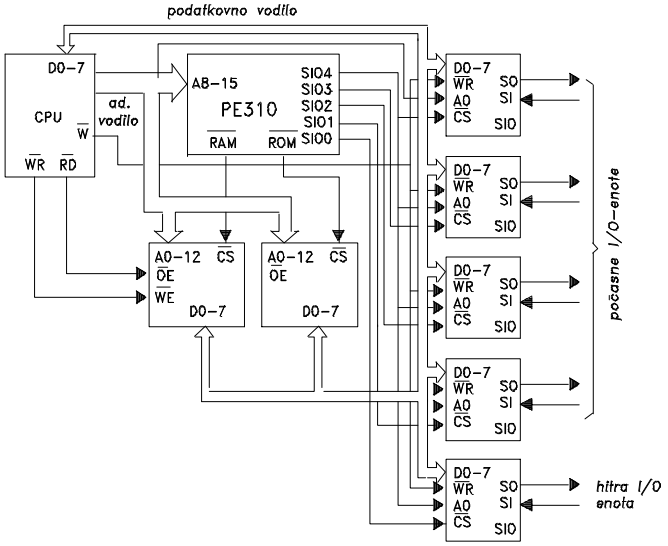
$$
\begin{aligned}
\mathrm{CS_{ROM}} &= \bar{A}_{15}\bar{A}_{14}\bar{A}_{13} \\
\mathrm{CS_{RAM}} &= \bar{A}_{15}\bar{A}_{14}A_{13} \\
\mathrm{CS_{SIO0}} &= A_{15}A_{14}A_{13}A_{12}A_{11}\bar{A}_{10}A_{9}\bar{A}_{8} \\
\mathrm{CS_{SIO1}} &= A_{15}A_{14}A_{13}A_{12}A_{11}\bar{A}_{10}A_{9}A_{8} \\
\mathrm{CS_{SIO2}} &= A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}\bar{A}_{9}\bar{A}_{8} \\
\mathrm{CS_{SIO3}} &= A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}\bar{A}_{9}A_{8} \\
\mathrm{CS_{SIO4}} &= A_{15}A_{14}A_{13}A_{12}A_{11}A_{10}A_{9}\bar{A}_{8}
\end{aligned}
$$

S temi funkcijami gremo v izvorno datoteko orodja A+PLUS (Altera Programming LogiC User System), ki nam po procesiranju da ustrezno JEDEC datoteko. A+PLUS obsega naslednje faze priprave EPLD-čipa:

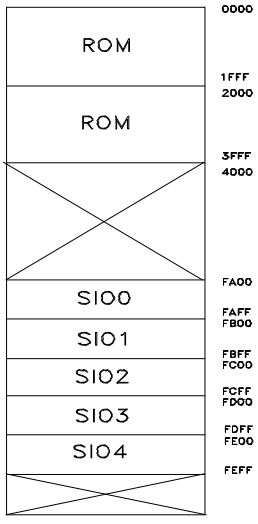
- zbirko Alterinih elementov
- zbirko Alterinih delov
- optimizacijo logičnega načrtovanja



Slika 6.2: Bločna shema čipa Altera PE900.



Slika 6.3: I/O-krmilnik mikročunalnika.



Slika 6.4: Pokritje naslovnega prostora.

- avtomatično označevanje elementov in priključkov
- kreiranje in lociranje čipa
- EPLD-prevajanje
- izdelavo JEDEC-datoteke
- verifikacijo
- "smart" programator
- EPLD-programator in
- korekcijske sposobnosti pri načrtovanju.

Vzemimo RS-pomnilno celico, ki je zgrajena z dvema NOR-operatorjema, ki imata vsak po dva vhoda. En vhod v obeh primerih je izkoriščen za povratno povezavo (doseganje pomnilne značilnosti), drugi vhod pa je uporabljen pri enem operatorju za vhod R in pri drugem za vhod S. Na izhodu celice je 3-stanjsko vezje, ki je v knjžnici primitivov označeno kot COCF. Krmiljenje 3-stanjskega vezja je zaupano signalu OUTCNTL. Izvorni program (v žargonu Altere design file) za takšno vezje je

```
ALTERA
xx/yy/1998
V.x
EP310
RS LATCH

PART: EP310
INPUTS: OUTCNTL, R,S
OUTPUTS: RSLATCH
NETWORK:
OUTCNTL   = INP(OUTCNTL)
R         = INP(R)
S         = INP(S)
RSLATCH,Q = COCF(RSLATCHc,OE)
EQUATIONS:
RSLATCHc=/(/(S+Q)+R);
OE = OUTCNTL;
END$
```

Vidimo, da pristop pri orodju A+PLUS ni prav nič drugačen kot na primer pri orodju ABEL, razlika je le v pogledu sintakse.

6.2.3 Brisanje EPLD-čipa

Rekli smo že, da lahko Altera-čipe zbrišemo in ponovno zapišemo. Brišemo s svetlobo valovne dolžine pod 4000 Å. Potrebno je povedati, da obstaja velika nevarnost za okvaro vsebine, če so čipi izpostavljeni sončni ali določeni fluorescenčni svetlobi. V teh svetlobah obstaja tudi valovna dolžina med 3000 in 4000 Å. Do izgube vsebine lahko pride, če so čipi izpostavljeni soncu npr. teden dni časa, medtem ko sobna fluorescenčna svetloba lahko povzroča težave po treh letih. Torej zatemnitev čipovega okna ni prazna beseda, saj se v nasprotnem primeru lahko znajdemo v silnih težavah. Najprimernejša valovna dolžina svetlobe za brisanje je 2530 Å. Intenzivnost te svetlobe, ki je potrebna za zanesljivo brisanje, je najmanj 15 Wsekcm^{-2}. Glede na to potrebujemo čas brisanja 15 do 20 min, če uporabljamo ultravioletno žarnico 1200 μW/cm^{-2}. Čip se naj nahaja palec od žarnice. Pri preveliki energiji osvetljevanja pride lahko do poškodbe - čipa ne bomo nikoli izpostavljali energiji nad 7000 Wsek/cm^{-2}! Prevelika intenzivnost lahko po določenemu času privede do trajnih poškodb.

Izrednega pomena je dejstvo, da so Altera-čipi vedno 100 % testirani. To pomeni, da je preverjen vsak EPROM-bit in vsak logični elemet na čipu. Med delovanjem morajo biti vsi prosti, neizkoriščeni priključki vedno vezani na določen potencial (Vcc, GND).

6.3 NAČRTOVANJE S XILINX LCA-TEHNIKO

6.3.1 Uvod

Firma Xilinx se je sistematično lotila dostopnosti k velikim logičnim razvrstitvam. Celice (logični bloki) so razvrščene koordinatno, kordinatno pa potekajo tudi programirne povezave med njimi. I/O-bloki so izrinjeni na navpične in vodoravne robove (v bližino priključkov). Tako postavljene strukture imenujejo LCA (logic cell arrays). Tudi LCA je realizirana s CMOS tehnologijo, kar da majhno porabo energije, relativno hitro delovanje in popolno testiranje čipov. Logične sheme in povezave med njimi v veliki meri koristijo programirne multipleksorje, ki dajo obilo možnih logičnih in povezovalnih kombinacij.

Xilinx izdeluje tako EPLD kot tudi FPGA-čipe. Notacija čipov je enaka, razmejitev pa je nekje pri številu 4000. Tako so čipi družine XC4000 še FPGA, medtem ko so čipi družine XC7000 EPLD-narave. FPGA uporablja

za osnovni proces CMOS-static RAM, medtem ko EPLD uporablja proces CMOS EPROM. EPLD-čipi se uporabljajo v bolj kompleksnih (kompleksni avtomati, multiport-krmilniki, akumulatorji, vmesniki med procesorjem in periferijo, aparaturne rešitve grafike itd.) in FPGA v manj kompleksnih shemah (enostavnejši avtomati, reprogramirne aplikacije, hitre števne naprave, vmesniki vodil itd.)

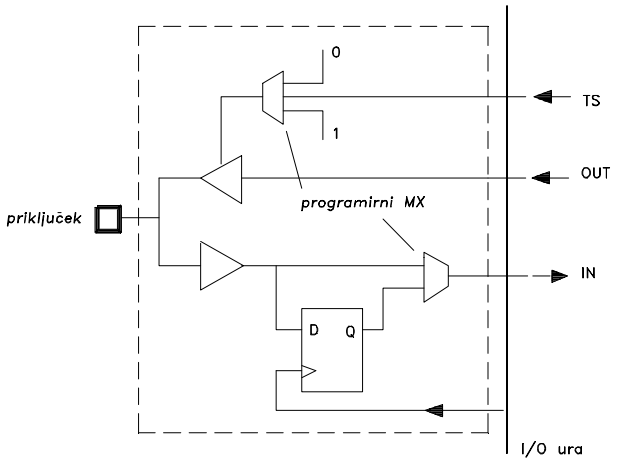
6.3.2 Osnovni bloki LC-razvrstitev

Temelj povezovanja in logike predstavlja statična CMOS-polnilna celica, ki je sestavljena iz dveh negatorjev, ki sta povratno sklopljena. V zanko je vpet dostopni CMOS-tranzistor ("pass" tranzistor), ki služi kot vhod in izhod za bit, shranjen v celici. V celico lahko vpisujemo samo pri postavljanju konfiguracije čipa. Tekom normalnega delovanja že programiranega čipa je dostopni tranzistor odprt, kar omogoča veliko stabilnost konfiguracije.

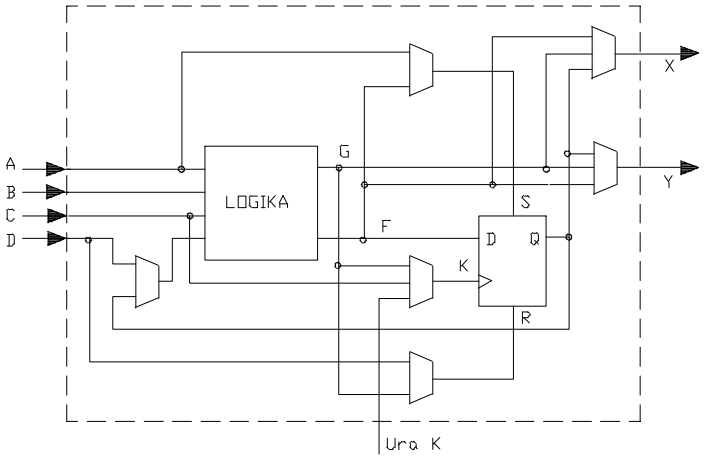
Vhodno-izhodni bloki omogočajo, da je na osnovi programiranja lahko vsak priključek ali vhod ali izhod. Tak blok vidimo na sliki 6.5. Signal $\overline{\mathsf{TS}}$ predstavlja dosegljivost izhoda OUT. Če je dosegljivost aktivna, ima OUT dostop do priključka. V nasprotnem primeru pride lahko signal IN iz priključka v notranjost LCA. Pri vhodu lahko odigra svojo vlogo tudi D-pomnilna celica, če je le spodnji programirni MX ustrezno programiran.

Konfiguracijski logični blok CLB (configurable logic block) za srednje velik čip vidimo na sliki 6.6. Iz njega izhaja, da je osrednja logična celica katerakoli logična funkcija štirih spremenljivk $f(A, B, C, D)$. Tudi v tem bloku funkcionalnost dokončno določi načrtovalec s programiranjem šest MX. Vsak logični blok ima LOGIKO, D-pomnilno celico in interno povezovanje (internal routing) omenjenih programirnih multipleksorjev. Izhoda X in Y sta vezana na povezovalno mrežo med bloki, kar omogoča nadgradnjo v smislu večje logične sheme. LOGIKA ni logično vezje, temveč tabela v pomnilniku, v kateri funkcijske vrednosti enostavno odčitavamo glede na vrednosti, ki jih imajo spremenljivke $A, B, C,$ in D. Namesto ene funkcije štirih spremenljivk f lahko generiramo dve funkciji, od katerih je vsaka odvisna od treh spremenljivk. Pri akciji sodeluje izhod Q D-celice, ki ga zato vodimo nazaj na vhodni MX. Tako pridemo do dveh neodvisnih funkcij F in G. Obstaja pa še tretji način delovanja, v katerem se spremenljivka B uporabi kot dinamična selekcijska spremenljika za generiranje nekaterih funkcij petih spremenljivk.

Na osnovi logičnega bloka na sliki 6.6 lahko programsko konfiguriramo



Slika 6.5: Vhodno-izhodni blok v LCA.



Slika 6.6: Konfiguracijski logični blok v LCA.

pomnilni blok enega bita (CLB storage element). V tem primeru pridejo v poštev osrednji programirni multipleksorji, ki jih vidimo na sliki 6.6. Takšni moduli na LCA omogočajo, da z LCA realiziramo pomnilnike, sekvenčna preklopna vezja in logične avtomate.

Med bloki CLB in IOB so potrebne ustrezne povezave. Povezovalna tehnika na Xilinx-čipih omogoča splošne, neposredne in dolge povezave. Splošne povezave upoštevajo 5 navpičnih in 5 vodoravnih povezovalnih segmentov, med katerimi se nahajajo bloki CLB in IOB. Učinkovitejša je neposredna povezava bližnjih blokov, ki ne koristi omemjenih splošnih povezovalnih možnosti, da pridemo do majhne propagacije. Dolge povezave so potrebne, da z njimi lahko obhajamo splošne povezovalne matrike. Obstajajo 3 takšne povezave na stolpec in 2 na vrstico blokov. Splošne povezovalne matrike se nahajajo na vsakem križišču navpičnih in vodoravnih vodil med bloki. Matrika je reda $n \times m$, če imamo v vodoravni smeri n vodil in v navpični smeri m vodil. Vsak blok ima povezovalno matriko nad sabo, pod sabo, na svoji levi in na desni strani. Zadnje omogoča segmentno gradnjo logične sheme. Če vežemo X ali Y na najbližji kanal vodila, pomeni, da ta zasede le najbližji segment kanala med naslednjo in prejšnjo povezovalno matriko. Na tak segmentni način s programiranjem povezovalne matrike gremo lahko od kateregakoli bloka do kateregakoli bloka (cik-cak po navpičnici ali po vodoravnici). Segmenti se lahko vežejo zaporedno (CLB je vezan na en naslednji CLB) ali paralelno (CLB je vezan na več naslenjih CLB).

Velikostni red LCA-čipov, ki ustreza sliki 6.6, predstavimo na primer s čipom XC2018. Ta obsega 1800 vrat, 100 konfiguracijskih blokov CLB, 74 priključkov, ki so lahko vhodi ali izhodi. Konfiguracijski program obsega 17878 bitov. Novejši čip XC3000 ima že večji CLB: z logiko $f(A, B, C, D, E)$, dvema D-pomnilnima celicama in osmimi programirnimi multipleksorji MX. Še večje in kompleksnejše CLB srečamo na čipih XC4000 in XC5200. Povečevanje Xilinx-čipa ne pomeni samo številčnega večanja enake logike, temveč se ta povečuje tudi po zmogljivosti oziroma sposobnosti.

6.3.3 Načrtovanje logičnih shem s LCA

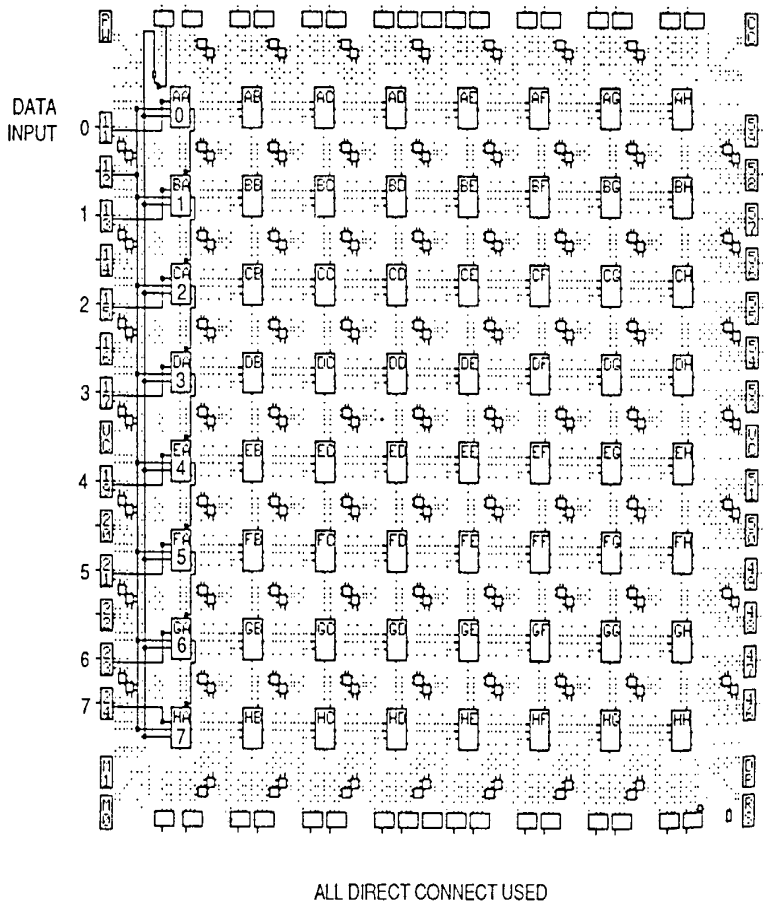
Iz prejšnjega razdelka izhaja, da se Xilinx LCA nanašajo na logični blok velikostnega reda katerekoli funkcije štirih spremenljivk $f(A, B, C, D)$ ali pa dveh neodvisnih funkcij treh spremenljivk. V kolikor je logična shema večja od omenjenih funkcij, je potrebna dekompozicija sheme na manjše dele tako, da se da vsak del posebej realizirati z enim CLB. Povezave med

deli (povezave dekompozicije) prevzamejo navpična in vodoravna vodila s svojimi programirnimi matrikami, vsebino v delih pa logični bloki s programirno sposobnostjo multipleksorjev MX. Pri omenjenih dekompozicijah pride prav znanje, ki smo ga podali v 3. poglavju. Tako na primer slika 3.2 prikazuje, kako prevedemo eno logično funkcijo $f(x_1, x_2, x_3, x_4, x_5)$ na pet funkcij $f(x_1, x_2, x_3)$. Prva funkcija v splošnem ni dosegljiva z enim obsežnim CLB, medtem ko je s petimi enostavnimi (tudi z manjšim številom) CLB dosegljiva.

LCA so zelo primerne za takšne logične sheme, ki kažejo na določeno iteracijo. Takšne sheme so na primer števne naprave, registri, pomnilniki itd. Čim večja je iteracija v logični shemi, bolj enostavni je programiranje povezovalnih matrik, ker se segmenti ne vežejo cik-cak, temveč ravno na večjo dolžino.

Na sliki 6.7 vidimo postavitev 8-bitnega premikalnega registra s paralelnimi vhodi [14], [25]. Namestitev je ob levi stranici, ostali prostor seveda lahko zasedajo logične sheme, ki so v bližini premikalnega registra. Uporabljene so neposredne povezave, ki smo jih že omenili. Primer nazorno prikazuje, da imamo lahko pri nameščeni logični shemi lahko dolge in kratke povezave. V naslednjem delu učbenika bomo videli, da ni vseeno ali je opazovana povezava kratka ali dolgo, če je le hitrost delovanja čipa visoka. Vidimo, da so vse povezave navpične in vodoravne. Na zadnji sliki si lahko pogledamo tudi, kako so označeni CLB, ki ležijo na vsej površini med koordinatnimi vodili.

Programiranje in postavljanje logičnih shem v LCA izvajamo na osnovi orodja XACT (Development system). To orodje vsebuje urejevalnik načrtovanja, vnašanje shemnih podatkov, simulacijo in emulacijo, računanje zakasnitev, časovno simulacijo in avtomatično polaganje logike in poti med logiko.



Slika 6.7: 8-bitni premikalni register s paralelnim vhodom.

Del III

UPOŠTEVANJE ČASA IN RAZDALJE V RSS

Poglavje 7

ČAS V NOTRANJOSTI LOGIČNE SHEME

7.1 UVOD

V že obravnavanih poglavjih se praktično nismo dotaknili časa kot parametra načrtovanja RSS. Ker sodi čas med najpomembnejše postavke načrtovanja, mu namenjamo poseben skupek poglavij. Podajanje logičnega načrtovanja RSS bi bilo zanesljivo prepomankljivo, če tudi času ne bi dali dovolj prostora. Na prvi pogled zgleda, da je čas povsem fizikalna postavka in naj bi bila zato stvar fizike, elektronike in mikroelektronike, ne pa tudi računalniškega načrtovanja. Pri malo globljem gledanju na stvar pa ugotovimo, da igra čas pomembno vlogo tudi pri logičnem načrtovanju.

Lahko rečemo, da ima čas pri načrtovanju RSS mnogo obrazov. Mnogo, pomeni, da lahko tem obrazom namenimo vso knjigo, pa bi še vedno ostalo mnogo nedotaknjenih obravnav časa. Kaj mislimo pod obrazi časa, vidimo v naslednjem spisku časov:

- globalni čas načrtovanja RSS,
- zakasnitev signala med dvema logičnima moduloma,
- zakasnitev signala preko decizijskega logičnega modula,
- hitrost delovanja načrtane RSS,
- logični čas v sekvenčnih vezjih, avtomatih
- dinamičnost signalov,
- časovna kontrola določenih akcij v RSS

  in še in še.

Sodobne RSS zahtevajo, da ima načrtovalec stalno pred očmi vlogo časa,

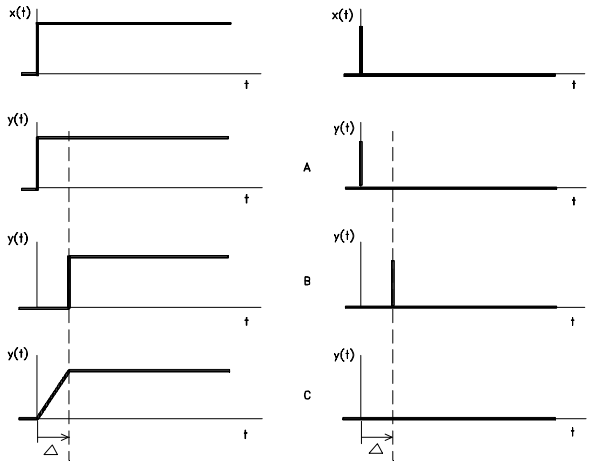
tudi če v načrtovani RSS ni pomnilnih struktur, ki imajo pomen za aplikacije. Tudi pri decizijskih logičnih shemah pride lahko do logike, ki je drugačna od želene, ali pa celo do divjih, nestabilnih delovanj, kar mora biti izključeno iz vsake RSS. Predvsem nas zanimajo časovne značilnosti in značilnosti razdalj signalov, ki se pretakajo v notranjosti logičnih shem in med takšnimi shemami.

Čas in razdalja v njegovi domeni v tem delu ne bi obravnavali, če ne bi bilo tržnega in tudi drugačnih pritiskov, ki zahtevajo, da se v RSS rapidno dviguje hitrost delovanja. V času, ko je PC deloval s hitrostjo 16 - 30 MHz računalnikarjem niso bila potrebna posebna znanja o internem delovanju RSS. Razmere so se povsem spremenile, saj današnji PC, WS in drugi množični sistemi delujejo pri osnovni hitrosti 133 - 300 MHz in več, pri vsem tem pa se RSS postavljajo s programirnimi čipi, ki dopuščajo vnašanje subjektivnosti v RSS.

7.2 ZAKASNITEV IN ZAMUDITEV VRAT

Logične funkcije v RSS zavisijo od preklopnih procesov in od pogojev pod katerimi ti procesi delujejo. Na sliki 7.1 vidimo preklopni proces v odvisnosti od časa. Ta proces je lahko stacionaren (levi stolpec signalov) ali dinamičen (desni stolpec signalov). x je vhodna in y izhodna logična spremenjivka logičnega vezja. V vrstici A imamo idealni časovni odgovor, ki ga lahko srečamo samo v idealnem logičnem vezju. Takšnega vezja v realnem svetu ni, ker, kakor hitro ga realiziramo, vpliva nanj fizikalno okolje, v katerem je čas bistvena postavka. Čas vpliva na signale na dva načina, katera sta predstavljena v vrsticah B in C. Zlasti je kritičen zadnji primer, na sliki desno spodaj, v katerem zaradi zakasnitve $\triangle$ celo izgubimo informacijo (zamuditev signala). Če informacijo izgubimo, je to za logično vezje zanesljivo nekaj drugega, kot bi sicer moralo biti.

Primer signalov na sliki 7.1 nazorno prikazuje, kakšen vpliv ima čas na logična vezja. Da lahko zajamemo v logiko tudi vpliv časa, moramo iz idealnih modelov preiti vsaj na realnejše modele (ne povsem realne, ker so realni modeli digitalno neobvladljivi) logičnih funkcij ali pa uvedemo sinhronizacijo z uro, kar podaljšuje status idealnega gledanja na logične sheme. Razliko med realnejšimi vrati in idealnimi vidimo na sliki 7.2. Funkcija f je rezultat idealnih vrat, medtem ko je y funkcija, ki nosi s seboj tudi vpliv časa $\triangle$. Spremenljivka Y je nemirnost logičnih vrat [1], ki jo moramo obvladati računsko sicer je naše načrtovanje RSS nepopolno. V splošnem vsaka realnejša logična vrata spremlja določena izhodna zakasnitev, ki jo



Slika 7.1: Vpliv časa na preklopni proces.

lahko po modelu realnejših vrat premaknemo na vse vhode opazovanih vrat. Realnejša vrata so enaka idealnim vratom, pred katerimi se na vhodih nahajajo vhodne zakasnitve, ki so enake omenjeni izhodni zakasnitvi. Prav tako je obsežnejša realnejša logična shema enaka idealni, ki ima na vsakem vhodu ustrezno vejno zakasnitev [1].

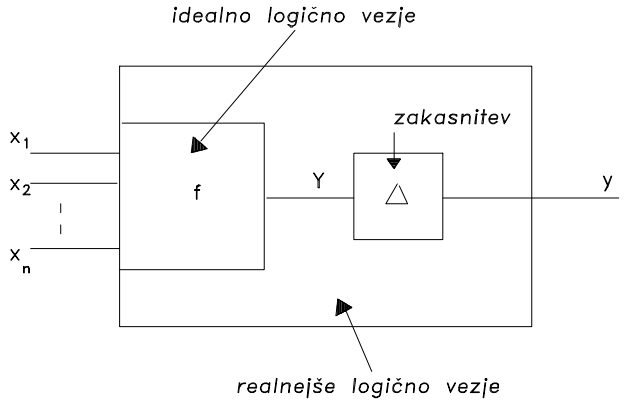
Po sliki 7.2 imamo v realnejših vratih

$$\begin{aligned} y(t) &= Y(t-\triangle) \quad \text{, če je } t \geqslant \triangle \\ y(t) &= y_0 \qquad\qquad \text{, če je } 0 \leqslant t < \triangle\,. \end{aligned} \tag{7.1}$$

y_0 je začetna vrednost izhoda. Na osnovnih odnosih (7.1) lahko vzpostavimo sistem logičnih enačb, ki opisuje delovanje logične sheme ob upoštevanju realnosti, ki jo nosi s seboj zakasnitev signalov.

7.3 ZAKASNITEV LOGIČNE SHEME

V logični shemi upoštevamo samo tiste zakasnitve, ki so znatne glede na hitrost njenega delovanja. Vzemimo, da polagamo logično shemo na sliki 7.3 na logično razporeditev. Da ne bomo logično shemo risali večkrat, vzemimo za začetek, da je $T_1 = T_2 = T_3 = T_4 = 0$. Pri tem privzetju idealne sheme imamo opraviti s funkcijo



Slika 7.2: Bločni odnos med realnejšimi in idealnimi vrati.

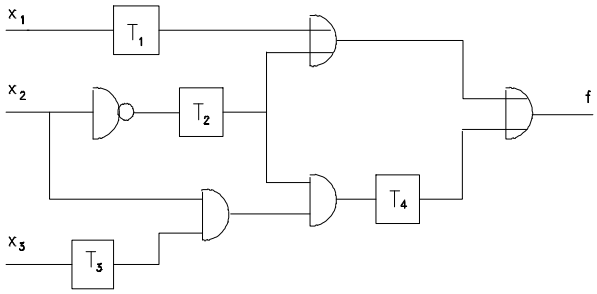
$$f(x_1, x_2, x_3) = x_1 \vee \bar{x}_2 \vee \bar{x}_2(x_2 x_3) = x_1 \vee \bar{x}_2. \tag{7.2}$$

Vidimo, da se shema zelo poenostavi, če jo lahko postavimo neodvisno od okolja v MDNO. Vendar iz določenih vzrokov moramo opazovano logično shemo postaviti tako, kot to prikazuje slika 7.3 s $T_a > 0$, $a = 1, 2, 3, 4$. Na sliki 6.7 smo spoznali, da pri polaganju oziroma namestitvi logične sheme na logično porazdelitev lahko obstajajo dolge in kratke povezave, zavisno, kako so si operatorji oziroma deli logične sheme med seboj oddaljeni. Pri naši realni postavitvi imamo takšne oddaljenosti, da ustreznih zakasnitev T_1, T_2, T_3 in T_4 ne moremo zanemariti. Kako jih upoštevamo in tudi formalno zapišemno?

Od vhodov x do izhoda f vodi toliko poti, kolikor je različnih vej v ustreznem grafu. Pri treh vhodih na sliki 7.3 imamo pet vej, vsaki pa pripada zakasnitev, ki je seštevek vseh T_a v veji. Tako imamo

$$\begin{aligned} k_{11} &= T_1 \\ k_{21} &= T_2 \\ k_{22} &= T_2 + T_4 \\ k_{23} &= T_4 \\ k_{31} &= T_3 + T_4\,. \end{aligned} \tag{7.3}$$

Indeks i časovne spremenljivke k_{ij} predstavlja i-ti vhod sheme in indeks j j-to vejo vhoda i. Za idealno logično shemo, ki ima na svojih vhodih k_{ij},



Slika 7.3: Logična shema, pri kateri je potrebno upoštevati zakasnitve T_1 do T_4.

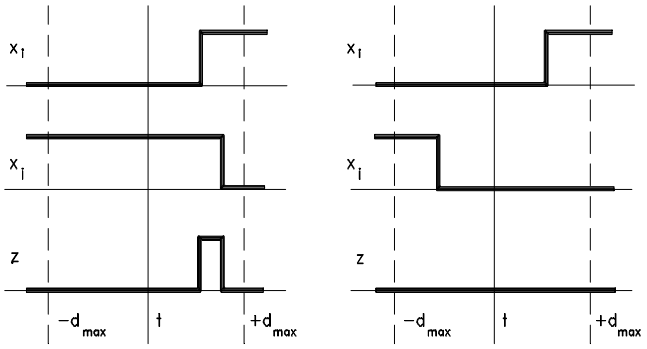
imamo sedaj spremenljivke x_{ij}, ki s seboj nosijo tudi vplivnost ustreznih zakasnitev. Namesto funkcije (7.2) imamo sedaj novo funkcijo, ki zavisi od spremenljivk x_{ij} oziroma od spremenljivk x_i in zakasnitev $\mathrm{D}^{-k_{ij}}$. Logično shemo na sliki 7.3 tako opredelimo s funkcijo

$$\begin{aligned} &f(x_{11}, x_{21}, x_{22}, x_{23}, x_{31}) = x_{11} \vee \bar{x}_{21} \vee \bar{x}_{22} x_{23} x_{31} = \\ &= \mathrm{D}^{-T_1} x_1 \vee \mathrm{D}^{-T_2} \bar{x}_2 \vee \mathrm{D}^{-T_2 - T_4} \bar{x}_2 \mathrm{D}^{-T_4} x_2 \mathrm{D}^{-T_3 - T_4} x_3 \,. \end{aligned} \tag{7.4}$$

Kako se s takimi logičnimi funkcijami, ki so, kot vidimo, odvisne tudi od časa, računamo, si lahko ogledamo v [1].

7.4 RAČUNANJE TRAVE V LOGIČNIH SHEMAH

Najprej moramo razčistiti, kaj ima trava sloh iskati v logičnih shemah. Travo nam bodo predstavljali dinamični signali, ki so posledica realnejših razmer (glede na idealne razmere) v logičnih shemah. Če smo odkriti, je taka trava v logičnih shemah lahko zelo koristna, lahko pa tudi škodljiva. Če značilnosti trave v logičnih shemah ne bi bilo, ne bi mogli postavljati pomnilnih celic in s tem tudi ne pomnilnikov [1]. Trava je škodljiva le v primeru, če po vseh pričakovanjih ta ne bi smela nastopiti, pa vendarle nastopi. Pri pomnjenju znamo z dinamičnimi signali sistematično računati, kaj pa je s takim izračunom v primeru, ko je trava škodljiva in nepričakovana?



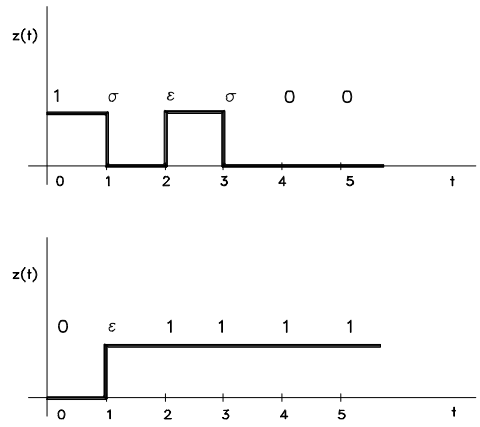
Slika 7.4: Realnejša preklopnost v logični shemi.

Na sliki 7.4 imamo časovni potek dveh preklopnih spremenljivk x_i in x_j v logični shemi. Zaradi hitre razlage naj bo ta shema kar konjunkcija $...x_i x_j...$. Spremenljivki x nista samo logično neodvisni, temveč tudi časovno. Ena spremenljivka je lahko blizu opazovanega mesta, druga pa daleč vstran. Zaradi zakasnitve vodil je preklopni proces obeh spremenljivk lahko tako premaknjen, da v določenem trenutku nastopi izhod $z = 1$. Pri drugačni namestitvi logične sheme pa do takšne vrednosti ne more priti. Trajanje motilne špice $z = 1$ (v splošnem 1- ali 0-špica signala) je lahko tako kratko, da je pri opazovanju s preprostim osciloskopom celo prezremo. Lahko pa traja dalj časa, odvisno od zakasnitev povezav in namestitev logičnih shem.

Kaj je s travo v logični shemi, lahko ugotovimo s testiranjem. V ta namen moramo vpeljati logiko, ki zna upoštevati tako stacionarne kot tudi dinamične signale (glej sliko 7.1). Namesto dvojiške logike z množico $\{0, 1\}$ vzamemo štiriško logiko z množico $\{0, 1, \sigma, \varepsilon\}$. Vrednost ε posnema dinamiko spremenljivke $0 \to 1$ in σ dinamiko $1 \to 0$. Logična spremenljivka tako lahko zavzame štiri vrednosti $x = 0$, $x = 1$, $x = 0 \to 1$ in $x = 1 \to 0$.

Vrnimo se k sliki 7.3 in opredelimo velikosti časovnih konstant T_i. Naj je $T_1 = T_2 = 1, T_3 = 3$ in $T_4 = 2$. Izberimo še testni vhodni vektor $\mathbf{x} = (0, \varepsilon, 1)$. Ta vektor pravi, da je vhodna spremenljivka x_1 stalno na vrednosti 0, spremenljivka x_3 je stalno na vrednosti 1 in spremenljivka x_2 se spreminja iz vrednosti 0 na vrednost 1.

Pri realno postavljenih časovnih konstantah nam enačba (7.4) da naslednje



Slika 7.5: Logični prehodni pojav pri vhodnem vektorju $\mathbf{x} = (0, \varepsilon, 1)$.

$$f(x_1, x_2, x_3) = \mathrm{D}^{-1}x_1 \vee \mathrm{D}^{-1}\bar{x}_2 \vee \mathrm{D}^{-3}\bar{x}_2\mathrm{D}^{-2}x_2\mathrm{D}^{-5}x_3\,. \qquad (7.5)$$

Pri vektorju $\mathbf{x} = (0, \varepsilon, 1)$ dobimo iz zadnjega izraza

$$z(0, \varepsilon, 1) = 0 \vee \mathrm{D}^{-1}\sigma \vee \mathrm{D}^{-3}\sigma\mathrm{D}^{-2}\varepsilon\,.$$

Temu odgovoru na izhodu logične sheme ustreza potek signala na sliki 7.5 zgoraj. Tega lahko zapišemo z niznim vektorjem $\mathbf{z}$(t)=(1,$\sigma, \varepsilon, \sigma, 0, 0, ...$). Torej, izhod logične sheme ne prehaja tako enostavno kot spremenljivka x_2 na vhodu sheme, saj se izhod ponovno vrne nazaj na vrednost 1 in se šele za tem umiri na vrednost 0.

Zgled 6 *:*

Dan je vhodni vektor $\mathbf{x} = (0, \sigma, 1)$. *Kako zgleda odziv izhoda* (7.5)?

Izhodni vektor je $\mathbf{z}(t) = (0, \varepsilon, 1, 1, ...)$. *Odziv na ta vektor vidimo na sliki* 7.5 *spodaj.*

Vse lepo in prav, če bi poznali relacijo med σ in ε. Vse odnose v množici $\{0, 1, \sigma, \varepsilon\}$ poznamo, neznan je le odnos med σ in ε. Ker je

temu tako, ne moremo določiti vrednosti za $\min(\sigma,\varepsilon)$, $\max(\sigma,\varepsilon)$ ali kakšne druge dvomestne logične operacije. Ker ne moremo zapisati nobene štiriške logične funkcije, ki bi predstavljala izhod s prehodnim pojavom, štiriško logiko razširimo na šestiško tako, da veljata relaciji

$$0 < \gamma < \varepsilon < \bar{\gamma} < 1 \qquad 0 < \gamma < \sigma < \bar{\gamma} < 1\,. \tag{7.6}$$

Možne vrednosti, ki jih sedaj lahko zavzame logična spremenljivka, so v množici $\{0, 1, \sigma, \varepsilon, \gamma, \bar{\gamma}\}$. Pri tem sta γ in $\bar{\gamma}$ takole opredeljeni

$$\mathrm{D}^{-k}\gamma = \mathrm{D}^{-k}\bar{x}\,\mathrm{D}^{-k}x \qquad \mathrm{D}^{-k}\bar{\gamma} = \mathrm{D}^{-k}\bar{x} \vee \mathrm{D}^{-k}x\,. \tag{7.7}$$

Vzemimo na sliki 7.4 $x_i = x$ in $x_j = \bar{x}$. Pri tem vidimo, kaj pravzaprav predstavlja vrednost γ. Ker je v tem primeru $z = \gamma$, je γ ozek impulz, ki se pojavi ali pa tudi ne, zavisno od tega, kje je nameščena spremenljivka x in kje $\bar{x}$. Če je ena na dolgi povezavi, druga pa na kratki povezavi, obstaja pri velikih hitrostih delovanja verjetnost za γ, ki je večja od 0.

Ob uvedbi γ in $\bar{\gamma}$ smo prišli do želenih relacij med σ in ε. Velja naslednje

$$\begin{aligned} \min(\sigma,\varepsilon) &= \min(\varepsilon,\sigma) = \gamma \\ \max(\sigma,\varepsilon) &= \max(\varepsilon,\sigma) = \bar{\gamma}\,. \end{aligned} \tag{7.8}$$

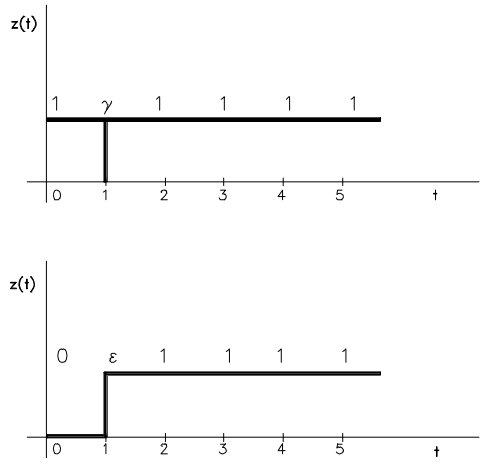
Če hočemo, da ne nastopi trava v opazovani logični shemi, mora v vseh primerih veljati

$$\begin{aligned} \min(\sigma,\varepsilon) &= \min(\varepsilon,\sigma) = 0 \\ \max(\sigma,\varepsilon) &= \max(\varepsilon,\sigma) = 1\,. \end{aligned} \tag{7.9}$$

Vrnimo se nekoliko nazaj k funciji (7.5). Pri vhodnem vektorju $\mathbf{x} = (\sigma,\sigma,\sigma)$ imamo v primeru šestiške logike naslednji izhod

$$\begin{gathered} z(\mathbf{x}) = \mathrm{D}^{-1}\sigma \vee \mathrm{D}^{-1}\varepsilon \vee \mathrm{D}^{-3}\varepsilon\mathrm{D}^{-5}\sigma\mathrm{D}^{-2}\sigma \\ \mathbf{z}(t) = (1,\gamma,1,1,1,...)\,. \end{gathered}$$

Ta vektor je prikazan na sliki 7.6 zgoraj. Pri času ene enote nastopi vrednost γ, povsod drugod pa imamo vrednost 1. Stacionarno stanje je v splošnem doseženo šele tedaj, ko pridemo pri vseh naslednjih časovnih enotah ali samo do vrednosti 1 ali samo do vrednosti 0.



Slika 7.6: Izhodni odziv logične sheme ob upoštevanju vrednosti γ in $\bar{\gamma}$.

Zgled 7 *:*

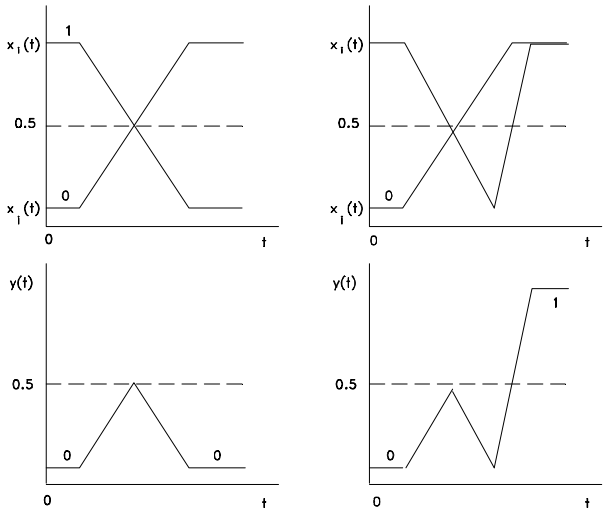
Kakšen je izhodni vektor, gledano seveda po času, če damo na vhod logične sheme s funkcijo (7.5) *vektor* $\mathbf{x} = (0, \sigma, \varepsilon)$**?**

$$z(\mathbf{x}) = 0 \vee \mathrm{D}^{-1}\varepsilon \vee \mathrm{D}^{-3}\varepsilon\mathrm{D}^{-5}\varepsilon\mathrm{D}^{-2}\sigma$$
$$\mathbf{z}(t) = (0, \varepsilon, 1, 1, 1, ...)\,.$$

Ustrezno sliko vektorja vidimo na sliki 7.6 *spodaj.*

7.5 LOGIČNI HAZARD

V predhodnem razdelku smo vpliv časa v logičnih shemah opazovali z vidika štiriške in šestiške logike. Če želimo realnejše opazovati logične sheme, se lahko poslužimo tudi teorije hazardnosti. Tudi pri hazardnosti nam je potrebno poseči po večvrednostni logiki, da pridemo do potrebnih razlag dogajanj. Za osnovo je privzeta trojiška logika, v kateri vsaka logična spremenljivka zavzame vrednost iz množice $\{0, \frac{1}{2}, 1\}$. Med 1 (ON) in 0 (OFF) vrednosti je tako vrinjena še vrednost $\frac{1}{2}$, ki nam pove, kaj je z logično shemo, če spremenljivke zavzemajo vrednosti, ki so med 0 in 1. Vrednost $x = \frac{1}{2}$ nam implicitno pove, da nastopa pri x sprememba $0 \rightarrow 1$ oziroma



Slika 7.7: Primerjava statičnega in dinamičnega hazarda pri konjunkciji dveh spremenljivk.

$1 \rightarrow 0$, kar smo v predhodnem razdelku označili z ε in σ. Pogoje, pod katerimi izhodi logičnih shem ne dajo takšnih vrednosti, ki jih zahtevajo spremembe vhodov, imenujemo hazard. 0-hazard imamo v primeru, če namesto 000 dobimo na izhodu 010 in 1-hazard, če namesto 111 dobimo na izhodu 101. V prvem primeru imamo 0X0 in v drugem 1X1, pri čemer nam X odkriva vrednost $\frac{1}{2}$. Če se spreminja samo ena vrednost, govorimo o statičnem hazardu. Pri takšnem hazardu je v trimestni časovni sekvenci prednja vrednost enaka zadnji vrednosti. Pri 0-hazardu je ta vrednost 0, pri 1-hazardu pa 1. Če se prednja in zadnja vrednost razlikujeta (prva je 0 in zadnja je 1 ali prva je 1 in zadnja je 0), govorimo o dinamičnem hazardu. Tako imamo lahko 01- ali 10-dinamični hazard. Razliko med statičnim in dinamičnim hazardom za $y = x_i x_j$ vidimo na sliki 7.7. Smiselno je, da zadnjo sliko primerjamo s sliko 7.4.

Za pogoj, da opazovana logična shema nima hazarda, lahko vzamemo kar izraza (7.9). Pri logičnem načrtovanju dobimo logične sheme brez hazarda, če vzamemo MDNO, ki vsebuje vse potrebne glavne vsebovalnike in k temu dodamo tudi vse nepotrebne glavne vsebovalnike, ki so nastopali pri postavitvi MDNO.

Logični hazard lahko opredelimo tudi z navadnim odvodom. Hazard h,

ki se nanaša na par vhodnih spremenljivk (x_1, x_2) in izhodno spremenljivko y, upošteva naslenje tri pogoje:

$$\begin{aligned} \dot{x}_1 \cdot \dot{x}_2 &= 1 \\ \frac{dy}{dx_1} \vee \frac{dy}{dx_2} &= 1 \\ \frac{d^2y}{d(x_1 \cdot x_2)} &= 0\,. \end{aligned} \tag{7.10}$$

V prvem izrazu nam $\dot{x}$ pomeni "časovni odvod" v smislu fronte signala. $\dot{x}(t) = x(t) \neq x(t+1)$. Velja $\dot{x}(t) = 0$, če je $x(t) = x(t+1)$. Prvi pogoj zahteva le to, da se v opazovanem času obe vhodni spremenljivki po vrednosti spreminjata. Drugi pogoj upošteva, da se y lahko spreminja ali po eni ali po drugi vhodni spremenljivki. Tretji pogoj pa opredeljuje ohranjanje vrednosti izhodne spremenljivke y. Hazard je tako podan z izrazom [16]

$$h = \dot{x}_1 \cdot \dot{x}_2 \cdot \left(\frac{dy}{dx_1} \vee \frac{dy}{dx_2} \right) \cdot \overline{\frac{d^2y}{d(x_1 \cdot x_2)}}\,. \tag{7.11}$$

Pri $h = 1$ je hazard, pri $h = 0$ ni hazarda. S tega vidika je hazard dvojiška spremenljivka.

7.6 SINHRONIZACIJA

7.6.1 Razlika med sinhronsko in asinhronsko rešitvijo RSS

V zanjem razdelku smo spoznali logični hazard, ki je lahko v RSS zelo nevaren, če se njegove odprave ne lotimo sistematično. Če hazardu ne posvetimo nobene ali dovolj pozornosti, realnejša (neidealna) logična vezja lahko vnesejo slučajnostne signale (travo), ki delovanje sekvenčnega vezja, avtomata pripelje do divjih sekvenc oziroma dogodkov, kar vodi v neuporabnost RSS. Rešitev opisanega problema lahko obvladamo na dva načina

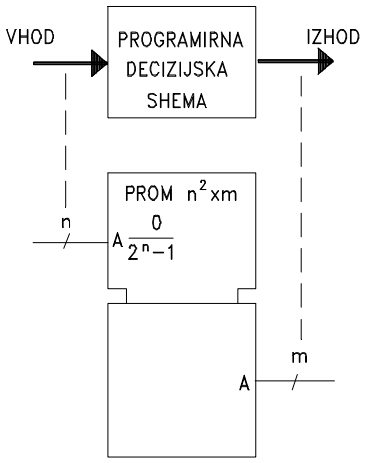
- znamo računati in predvidevati logični hazard (asinhronske rešitve RSS) in
- iz realnejše logike preidemo na idealno s pomočjo sinhronizacije (sinhronske rešitve RSS).

Zakaj je zanimiva prva točka, če obstaja druga, ki jo je na vrhu vsega zelo enostavno izpeljati? Asinhronske rešitve so časovno in informacijsko učinkovite, ker so osnovane na dinamičnosti signalov, medtem ko imamo v sinhronskih rešitvah opraviti s stacionarnimi signali (če izvzamemo urno delovanje). Asinhronska naprava reagira takoj, ko se pojavi na njej sprememba vhodnega signala. Sinhronska naprava pa čaka toliko časa, da pride do umeritve signalov (pridejo v stacionarno stanje). Ko so vsi signali v stacionarnem stanju, se dogodi urni impulz, ki povzroči v napravi potrebne spremembe. Med fronto vhodnega signala in urnim signalom poteče določen čas t, ki se v dobi delovanja naprave akumulira in ta akumulirani čas v principu predstavlja izgubljeni čas, saj je za uporabnika naprave, RSS, nedosegljiv, saj ga sistem potrebuje sam zase, zaradi pravilnega delovanja.

Asinhroske rešitve so informacijsko polnejše od svojih sinhronskih inačic. Vzemimo na primer dva dvojiška kanala, ki vstopata v računalniško napravo. V sinhronskem primeru imamo tedaj vhodno abecedo $X = \{00, 01, 10, 11)$, torej v njej so 4 črke. V asinhronskem primeru naprave imamo dinamično vhodno abecedo s črkami

$$00 \to 01, 00 \to 10, \underline{00 \to 11},$$
$$01 \to 00, \underline{01 \to 10}, 01 \to 11$$
$$10 \to 00, \underline{10 \to 01}, 10 \to 11,$$
$$\underline{11 \to 00}, 11 \to 01, 11 \to 10.$$

Prišli smo do 12 črk v dinamični abecedi, kar je 3 krat več od stacionarnega primera X. Podčrtane dinamične črke so tiste, ki lahko vnesejo velike hazardnosti v RSS zato, ker se v opazovanem trenutku spreminjata dve vrednosti hkrati (glej sliko 7.4). Tudi če 3 črke z dvojnim prehodom odštejemo, v dinamični abecedi še vedno ostane 9 črk, kar je precej več od 4 v abecedi X. Če imamo v RSS pomanjkanje kanalov glede na potrebno informacijsko količino, je dobro, da uberemo asinhronsko rešitev. Tipični primer pomanjkanja informacijskih kanalov je na primer v klasični telefoniji. Ni vseeno ali pri telefoniranju iz LJ v NY uporabimo en kanal (parico) ali tri ali pet kanalov (paric). Telefonska parica prevzame uporabnikove informacije in hkrati tudi vse sistemske (kontrolne) informacije, ki so potrebne, da se vzpostavi in prekine informacijska pot med LJ in NY. Vidimo, da je v klasični telefoniji velika racionalnost informacijkih poti. Prav zato se še vedno v praksi postavlja, čeprav imamo po drugi strani visoko infromacijko tehnologijo, ki poskuša omenjeno racionalnost nadoknaditi z drugimi in drugačnimi učinki.



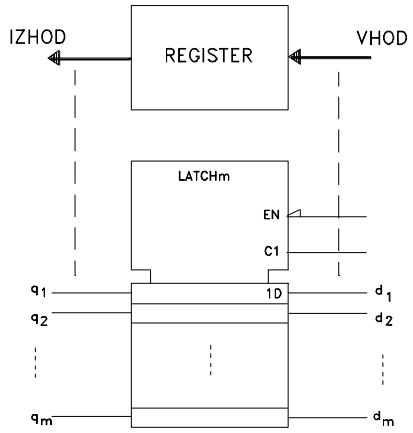
Slika 7.8: Odločitveno vezje podano s splošnim IEC-simbolom programirnega gradnika.

7.6.2 IEC-predstavitev decizije in sekvenčnosti

V naslednjih razdelkih želimo prikazati princip sinhronizacije v okolju programirnih komponent. Da bomo to lahko napravili, si najprej poglejmo, kako z IEC-simboli predstavimo programirno odločitveno shemo. Na slliki 7.8 vidimo PROM s svojim kontrolnim delom, ki predvsem obsega vhode naslavljanja A. Če damo na vhod naslov a, dobimo na izhodu vsebino $f(a)$, ki je shranjena pod tem naslovom. To vsebino predhodno programiramo skladno s pravilnostno tabelo odločitvene sheme. Seveda lahko namesto PROM vzamemo EPROM, EEPR0M, ROM; važno je le, da ima gradnik možnost programiranja s strani načrtovalca RSS.

Podobno lahko izrazimo tudi sekvenčno shemo. V tem primeru moramo izbrati registrski gradnik ali, pomnilnik, ki ga uporabimo kot register. Register, ki ga vidimo na sliki 7.9, je grajen z D-pomnilnimi celicami, s krmilno funkcijo $d = \mathrm{D}^1 q$. V primeru asinhronskega načina dela registra vzamemo $\mathsf{C1} = \mathsf{LE}$ (latch). Pri tem pa velja pravilnostna tabela

$\overline{\mathsf{EN}}$	LE	d_i	notranji izhod	zunanji izhod q
L	H	L	L	L
L	H	H	H	H



Slika 7.9: Realizacija pomnilnega registra s pomnilnikom RAM.

Pri C1=LE= H se izhod q_i obnaša v času $t+1$ tako, kot vhod d_i v času t. Prehod iz časa t v čas $t+1$ določa sprememba na vhodu d_i.

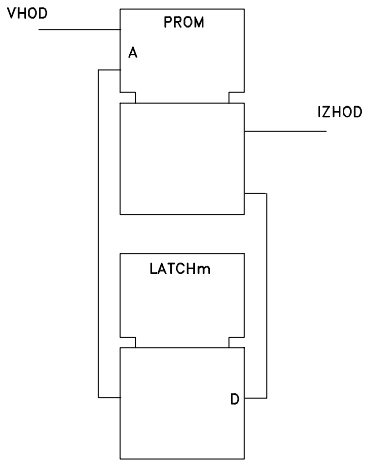
V primeru da je C1 spremenljivka, imamo pred seboj urni vhod za sinhronizacijo. Ta spremenljivka je nujno potrebna, ko postavljamo sinhronske rešitve RSS.

7.6.3 Asinhronska rešitev sekvenčne sheme

Na sliki 7.10 zgoraj imamo programirno odločitveno shemo, spodaj pa register z zatiči. Ti niso odvisni od nobenega urnega signala tako, da, če se spremeni VHOD, se spremeni tudi IZHOD. Primarni in sekundarni vhodi tvorijo skupaj naslov, pod katerim se nahaja tako izhod, kot naslednje stanje sekvenčne logične sheme. Pri asinhronski rešitvi je bistveno, da ni izkoriščen v smislu dinamičnega dogajanja v shemi noben vhod C, če se le-ta nahaja na uporabljenem gradniku. Takšen vhod, če že obstaja, mora biti na logični vrednosti 1 ali 0, zavisno od vrste gradnika.

7.6.4 Sinhronske rešitve sekvenčne sheme

Da v logični shemi preprečimo kakršenkoli hazard, ki bi ga povzročile pomnilne celice, uporabimo urni vhod C kot logično spremenljivko (C ni konstanta). Mesto sinhronizacije je le na vhodu pomnilnih celic, če je na shemi



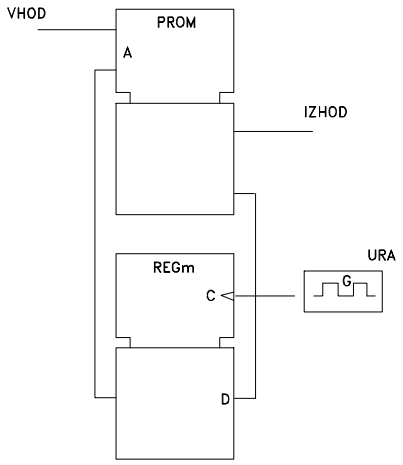
Slika 7.10: Asinhronska rešitev sekvenčne sheme.

VHOD, ki je že vsklajen, ne s katerokoli uro, temveč prav z uro C (slika 7.11).

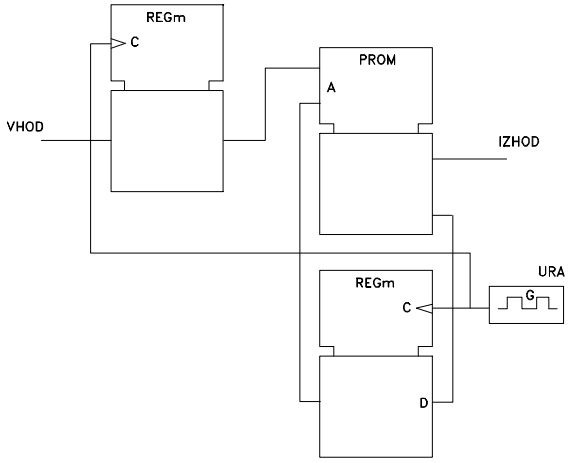
Če so vhodi (VHOD) asinhronski, je potrebno tudi te sinhronizirati na isti urni signal, kot je to v primeru sinhronizacije stanj sekvenčne sheme (slika 7.12).

V praksi se uveljavlja sinhronizacija, ki deluje na prednjo in zadnjo fronto urnega signala. Tak primer sinhronizacije imamo na primer pri pomnilni celici s predpomnjenjem ('master-slave' celica) [1]. V kolikor uvedemo takšno sinhronizacijo (two level-sensitive sinchronization) v našo sekvenčno shemo na vhodni strani, imamo opraviti s sliko 7.13. Prvi gradnik na vhodu se sinhronizira glede na prvo in drugi na drugo fronto urnega signala C1. Izhod prvega je neposredno vezan na vhod drugega vhodnega gradnika.

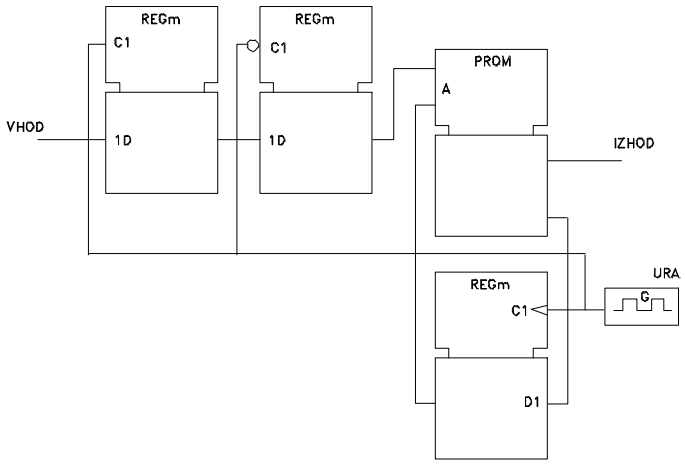
Večkrat diagram prehajanja stanj nima tipičnega vračanja stanja nazaj v predhodna stanja, ker se ta nadaljujejo v smislu vejanja drevesa (iz debla na veje in/ali iz vej na deblo). V tem primeru je praktično, da namesto registra v sekvenčni shemi izberemo programirno števno napravo. Na sliki 7.14 vidimo tak diagram prehajanja stanj. Sekvenčno shemo, ki je v tem primeru primerna, vidimo na sliki 7.15. Poleg decizijske sheme imamo v realizaciji le še števno enoto CRT. Ta ima paralelne vhode in praralelne izhode. Če je $M1 = 1$, števna naprava šteje in če je $M1 = 0$, da na svojem izhodu vhodne vrednosti. Takšno delovanje števne enote lahko inhibiramo



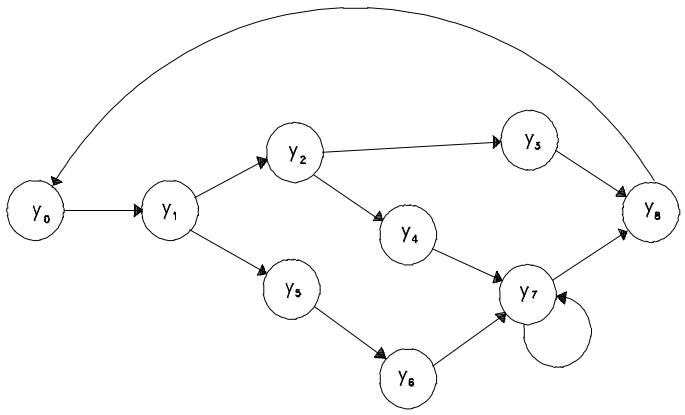
Slika 7.11: Sinhronizacija stanj sekvenčne sheme.



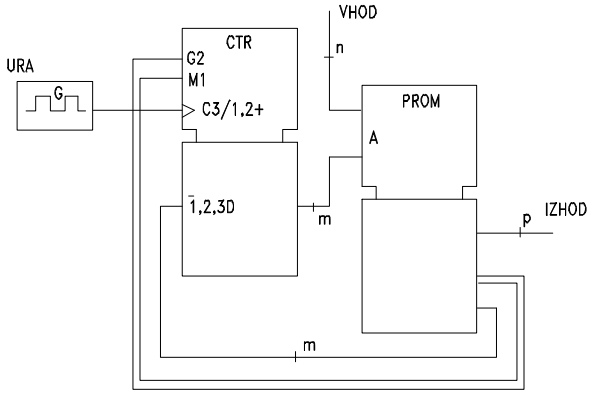
Slika 7.12: Sinhronizacija stanj sekvenčne sheme.



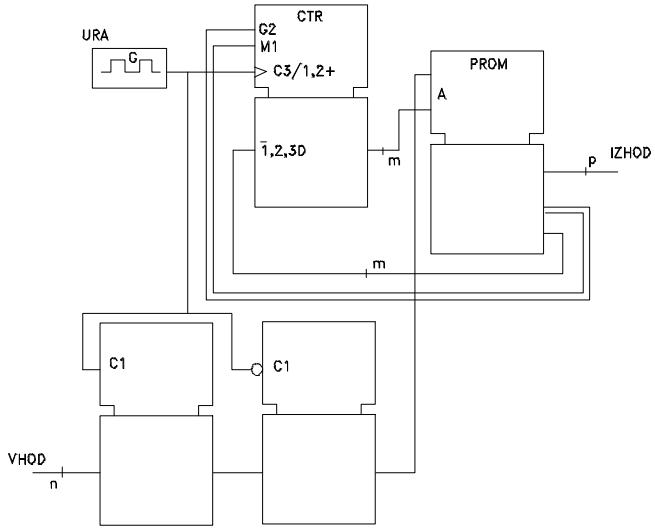
Slika 7.13: Frontna sinhronizacija vhodov programirne sekvenčne sheme.



Slika 7.14: Diagram prehajanja stanj, primeren za števno sekvenčno realizacijo.



Slika 7.15: Števna realizacija sekvenčne sheme.



Slika 7.16: Sinhronizacija v števni sekvenčni shemi.

z vrednostjo G2 = 0. Števne sheme uporabljamo zlasti v primeru logičnih kontrolerjev.

Sekvenčna shema 7.15 ima uro C3, ki jo za svoje delo potrebuje števna enota. Če so vhodi decizijske sheme od te ure neodvisni, jih je potrebno sinhronizirati, podobno, kot smo to napravili že pri registrski shemi. Eno od možnosti vidimo na sliki 7.16.

Poglavje 8

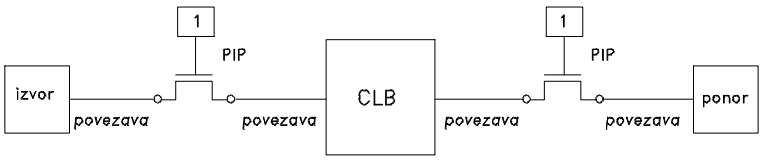
ZAKASNITEV POVEZAV

8.1 UVOD

V prejšnjem poglavju smo si ogledali bistvene časovne posebnosti v notranjosti logične sheme. V tem poglavju pa vzemimo logično shemo kot črno škatlo in opazujemo le vhode in izhod (izhode) sheme. Imamo dve možnosti: da na osnovi izhodnega signala gradimo še večje logične sheme (črne škatle) ali pa smo prišli že do izhoda RSS. Med logičnimi shemami so lahko razdalje kratke (vse v notranjosti enega čipa), lahko pa tudi dolge med različnimi čipi ali celo različnimi RSS. V primeru daljših povezav in visoke frekvence ure moramo jemati povezavo kot prenosno linijo, z vsemi značilnimi prenosnimi problemi. Povezava je lahko dolga že v domeni enega čipa, če je le frekvenca ure zelo visoka in je načrtovalcu dana programirna možnost, v okviru katere lahko postavlja dolge povezave.

8.2 LCA-POVEZAVA

Če se še enkrat ozremo na primer na sliko 6.7, vidimo, da lahko obstajajo daljše in krajše povezave med logičnimi bloki. Pri kratki povezavi je zakasnitev signala majhna, pri dolgi je zakasnitev večja in lahko zelo vplivna. Med seboj povezani logični bloki morajo delovati uskaljeno, vsi kot ena logična celota - gledano tudi na čas dogajanj v LCA-vezju. Z urnim signalom sinhroniziramo marsikaj, vendar ostane še vedno vrsta asinhronskih odnosov, ki morajo biti med seboj usklajeni. Dobro orodje za načrtovanje RSS avtomatično opravlja tudi časovno analizo logičnih shem (na primer za FPGA, LCA orodje XACT).

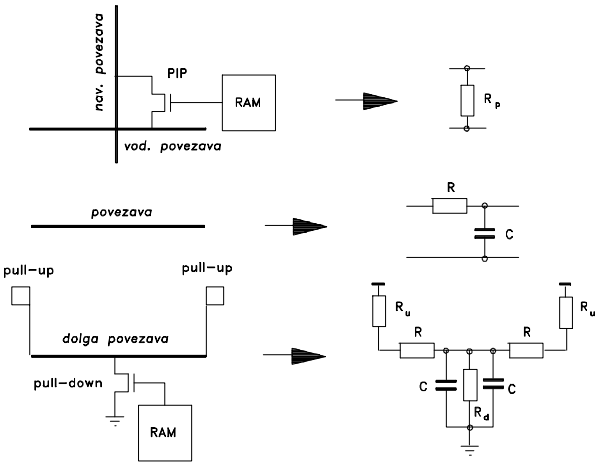


Slika 8.1: Zakasnitev segmentov povezav za CLB.

Ne da bi izgubili na splošnosti, si poglejmo povezavo izvor signala - CLB - ponor signala na sliki 8.1. PIP so programirne povezovalne točke (programmable interconnection points), ki povežejo CLB na povezovalne segmente 'povezava'. Oznaka 'povezava' na sliki pomeni fizično metalno pot med izvorom in PIP, med PIP in CLB, med CLB in PIP ter PIP in ponor signala. Ponor in izvor sta lahko priključka (preko I/O-bloka) ali pa drugi CLB v opazovani logični shemi.

V povezovalni matriki povezujemo vodoravni in navpični povezovalni segment, kot to vidimo na sliki 8.2. PIP-točka nam v svojem stacionarnem stanju predstavlja ohmski upor R_p. V sredini slike je podan segment povezave, ki je modeliran z vidika časa s konstanto RC. Ker ima metalizirani segment precej manjši ohmski upor od R_p, se večkrat R pri časovnih analizah opušča. Dolg segment je navadno vpet med "pull-up" točki in "pull-down" točko, kot to vidimo v spodnjem delu slike 8.2. Na desni strani vidimo RC-model za takšno stanje segmenta.

Obstaja ocena, da od 100 % zakasnitve, ki jo ima logika skupaj s povezavo, odpade 40 do 60 % na povezovanje. Zadnje je vzrok, zakaj pri načrtovanju ne moremo ignorirati zakasnitve povezave (net dalay). Zakaj je zakasnitev povezave stvar računalnikarja in ne mikroelektronika? Mikroelektronik pri proizvodnji čipa lahko predvidi le stalno zakasnitev I/O-bloka in CLB. Te zakasnitve se pri programiranju, ki ga opravlja računalnikar, načrtovalec RSS, ne spreminjajo, zato jim tudi rečemo stalne zakasnitve. Mikroelektronik pri proizvodnji čipa ne more upoštevati vse eksotike povezav, ki si jih lahko domislijo načrtovalci RSS. Daljši ali krajši povezavi ustreza večja ali manjša zakasnitev (gibka zakasnitev), o kateri lahko razmišlja samo še računalnikar, saj se čip nahaja že v njegovem laboratoriju, v katerem nima več kontakta s proizvajalcem.



Slika 8.2: Nadomestna RC-vezja za računanje zakasnitev povezovanja v LCA.

8.3 ELMOREOVA ZAKASNITEV

Glede na model v spodnjem delu slike 8.2 si lahko sami izračunavamo zakasnitve povezav oziroma za nas to opravlja računalniško orodje za načrtovanje. Na sliki 8.3 vidimo povezavo treh segmentov. Zgoraj imamo zaporedno vezavo segmentov, v sredini ustrezno nadomestno elektronsko vezje, spodaj pa je nadomestno vezje za paralelno delovanje na dva segmenta (CBL krmili paralelno dva nadaljnja CBL). Naloga, ki je pred nami, je, da določimo zakasnitev vsakega spojišča oziroma segmentnega dela posebej.

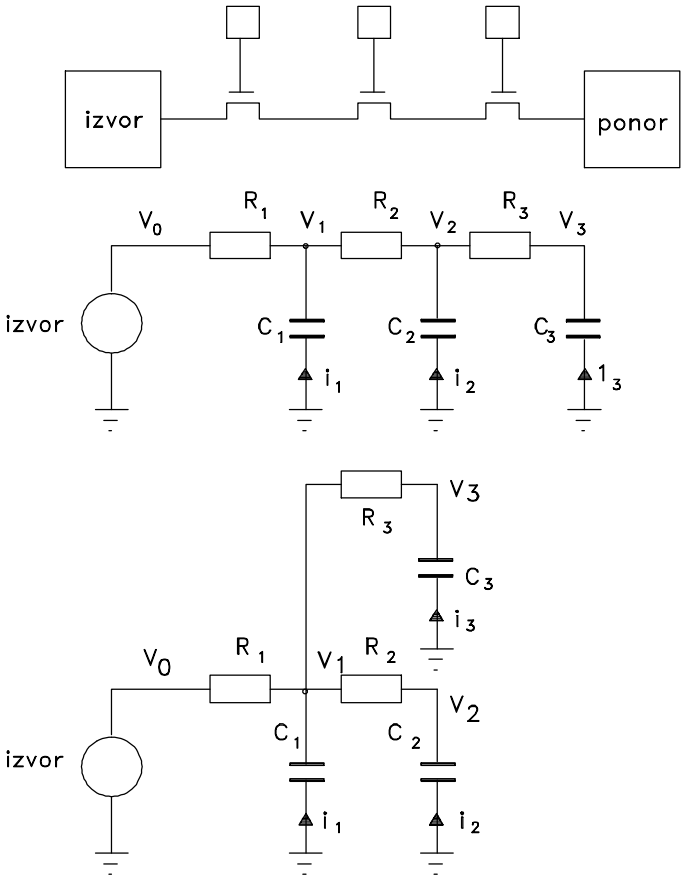
V splošnem ima nadomestno vezje spojišča i in veje k, ki ležijo med temi spojišči. V veji k teče tok i_k

$$i_k = -C_k \frac{dV_k}{dt}, \tag{8.1}$$

napetost spojišča i pa je

$$V_i = -\sum_{k=1}^{n} R_{ki} C_k \frac{dV_k}{dv} \tag{8.2}$$

R_{ki} je vzdolžna upornost v vejah vezja $V_0 \rightarrow V_k$ in $V_0 \rightarrow V_i$. Za gornji



Slika 8.3: Dve verigi RC-časovnih konstant povezovanja v LCA.

primer na sliki 8.3 imamo matriko vzdolžnih uporov

$$\mathbf{R} = \begin{bmatrix} R_1 & R_1 & R_1 \\ R_1 & R_1 + R_2 & R_1 + R_2 \\ R_1 & R_1 + R_2 & R_1 + R_2 + R_3 \end{bmatrix} \tag{8.3}$$

Poleg matrike $\mathbf{R}$ imamo vector kapacitivnosti $\mathbf{c} = [C_1, C_2, C_3]$. Vse zakasnitve vseh spojišč nadomestne sheme (segmentov povezave) dobimo s produktom

$$\mathbf{d} = \mathbf{Rc} = \begin{bmatrix} R_1 & R_1 & R_1 \\ R_1 & R_1 + R_2 & R_1 + R_2 \\ R_1 & R_1 + R_2 & R_1 + R_2 + R_3 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix} \tag{8.4}$$

$$\begin{aligned} d_1 &= R_1(C_1 + C_2 + C_3) \\ d_2 &= d_1 + R_2(C_2 + C_3) \\ d_3 &= d_2 + R_3 C_3 \,. \end{aligned} \tag{8.5}$$

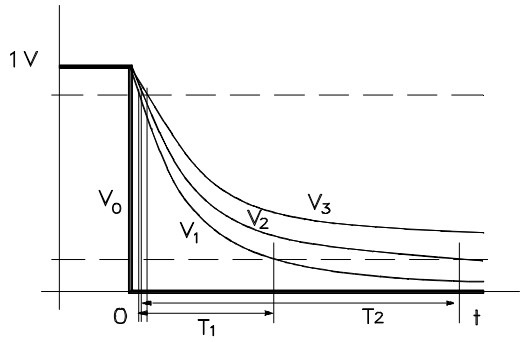
Zgled 8 *:*

Za spodnjo nadomestno shemo na sliki 8.3 *določimo zakasnitve segmentov povezave.*

$$\mathbf{d} = \mathbf{Rc} = \begin{bmatrix} R_1 & R_1 & R_1 \\ R_1 & R_1 + R_2 & R_1 \\ R_1 & R_1 & R_1 + R_3 \end{bmatrix} \begin{bmatrix} C_1 \\ C_2 \\ C_3 \end{bmatrix}$$

$$\begin{aligned} d_1 &= R_1(C_1 + C_2 + C_3) \\ d_2 &= d_1 + R_2 C_2 \\ d_3 &= d_1 + R_3 C_3 \,. \end{aligned}$$

V zgornjih izrazih obravnavo zakasnitve zapuščamo na nivoju posameznih časovnih konstant, ki določajo zakasnitev signalov. V končni meri nas zanima zakasnitev signala med začetno točko 0 in določeno končno točko i z napetostjo V_i in konstanto d_i. Če ima napetost V_i eksponentni potek, je zakasnitev enaka Elmoreovi časovni konstanti d_i pri upoštevanju mejnih točk $V_{0\max}/V_{1\min}$ poteka signala. $V_{0\max}$ je zgornja meja logične vrednosti 0 in $V_{1\min}$ spodnja meja signala 1. Načelni prikaz tako podanih napetosti V_i



Slika 8.4: Primer poteka napetosti spojišč, ko se zapre stikalo na vhodu vezja.

za primer na sliki 8.3 vidimo na sliki 8.4. V tem primeru imamo za Izvor stikalo, ki se pri napetosti 1 V sklene tako, da postane $V_0 = 0$ v času $t = 0$.

Potrebno je še povedati, da je Elmoreova zakasnitev v elektroniki splošno znana postavka, tako da ta ni nekaj posebnega samo pri analizi LCA-vezij, v okviru katere jo mi omenjamo.

Poglavje 9

ČAS IN RAZDALJE MED LOGIČNIMI SHEMAMI

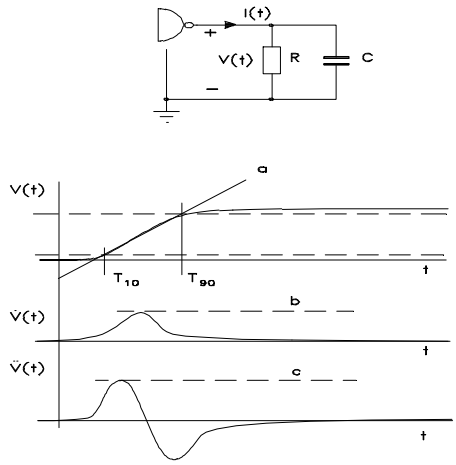
9.1 RAZMERE NA IZHODU LOGIČNE SHEME

Procesiranje v RSS zahteva signale in pretakanje le-teh med podsestavi. Hitrost pretakanja zavisi od osnovne ure in od medija, v katerem se pretakanje signalov dogaja. Tako imamo zakasnitve:

- ▷ pri koaksialnem kablu (dielektrična konstanta 1.8) 45 ps/cm
- ▷ pri koaksialnem kablu (dielektrična konstanta 2.3) 51 ps/cm
- ▷ pri povezavah na 4-stranski kartici, zunanja stran (dielektrična konstanta 2.8-4.5) 55-71 ps/cm
- ▷ pri povezavah na 4-stranski kartici, notranja stran (dielektrična konstanta 4.5) 71 ps/cm
- ▷ na velikostranskih Alumina-karticah (keramika, do 50 strani) pri povezavah na notranji strani (dielektrična konstanta 8-10) 94-106 ps/cm
- ▷ razširjanje po zraku (radijsko valovanje) (dielektrična konstanta 1) 34 ps/cm.

Pri procesorju z uro 330 MHz je dolžina urnega cikla 3000 ps. Tako je na primer pri dolžini povezave 30 cm na velikostranski Alumina-kartici na pohodu že naslednji impulz, ko opazovani še ni opravil svoje funkcije.

Stopničasti potek signala ima zaradi RC konstant medija čas naraščanja signala T_r in seveda tudi čas padanja signala T_f. Dovolj je, da si nadrobneje ogledamo le T_r. Med dolžino oziroma časom dvigovanja in zakasnitvijo medija velja odnos



Slika 9.1: Časovna dogajanja na izhodu logičnih vrat pri spremembi napetosti $\triangle V$.

$$L = \frac{T_r}{D} \tag{9.1}$$

Običajne mere pri tem so: L v cm, zakasnitev D v ps/cm in čas naraščanja T_r v ps. Čas naraščanja signala se računa od 10 % (še 0) do 90 % (že 1). Sprememba napetosti po času se tako računa kot

$$\frac{dV}{dt} \simeq \frac{\triangle V}{T_{10-90}}. \tag{9.2}$$

Največkrat nam logična vrata krmilijo naslednja vrata ali več naslednjih vrat, katere nadomestimo z nadomestnim RC-vezjem na sliki 9.1 zgoraj. Izhodni signal mora dovolj hitro premagati inercijo na svojem izhodu, ki jo opazujemo z vidika električnega toka $I(t)$ in električne napetodsti $V(t)$. Veljajo odnosi in časovni poteki, ki so podani v spodnjem delu slike 9.1. Tako imamo

$$\frac{dI(t)}{dt} = \frac{V(t)}{R} + \frac{dV(t)}{dt} C\,. \tag{9.3}$$

$$\begin{aligned} &\text{sprememba napetosti: } \triangle V = V(90) - V(10) \\ &b \text{ oz. naklon od } a\text{: } \frac{dV(t)}{dt} \simeq \frac{\triangle V}{T_{10-90}} \\ &c : \frac{d^2V(t)}{dt^2} \simeq 1.52\ \frac{\triangle V}{T_{10-90}^2}\,. \end{aligned} \tag{9.4}$$

Zgled 9 *:*

Kakšna je največja sprememba vrednosti toka skozi upor $R = 100$ - *in skozi kondenzator* $C = 50$ pF, *če je na izhodu* TTL-*vrat* $\triangle V = 3.7$ V *pri* $T_r = 3$ ns?

$$\frac{dI_R}{dt} = \frac{\triangle V}{T_{10-90}}\frac{1}{R} = \frac{3.7}{3\ 10^{-9}\ \cdot 100} \simeq 1.23\ 10^7\ \text{A/s}$$

$$\frac{dI_R}{dt} = \frac{1.52 \cdot \triangle V \cdot C}{T_{10-90}^2} = \frac{1.52 \cdot 3.7 \cdot 50\ 10^{-9}}{9\ 10^{-18}\ \cdot 100} \simeq 31.34\ 10^7\ \text{A/s}\,.$$

Vidimo, da so spremebe tokov po času zelo velike. Te lahko povzročajo dokaj nepričakovane težave (na primer presluha med povezavami, radijske motnje s strani kartice itd.).

9.2 PRENOSNA LINIJA

Dolga povezava med čipi, dolga povezava na čipih, vsak kanal adresnega ali podatkovnega vodila itd. je v principu prenosna linija. Glede na to je dobro, da se seznanimo tudi z osnovnimi performansami takšne linije.

Vsaka prenosna linija ima svojo karakteristično impedanco $Z_0 = \sqrt{L/C}$, svoj zaključni člen (gledano iz linije ven, lahko kombinacija R_0, C_0) in na vhodni strani navadno vhodni upor R_1 (gledano iz linije ven, notranja upornost izvora signala). Signal se razširja po liniji s določenim zakasnilnim časom (propagation delay, time of flight) t_f. Če linija ni prilagojena, so na koncu odboji, ki se razširjajo v nasprotni smeri, tako da je prehodni pojav lahko končan šele v času $2 \times t_f$, ker rabi signal čas za razširjanje naprej in

za nazaj po istem mediju. Velikost vala, ki se razširja po liniji doseže vrednost $Z_0/(R_1 + Z_0)$. Če je $R_1 = Z_0$, val doseže polovico velikosti vhodnega signala. Tipična karakteristična impedanca povezave na tiskanini je 50 do 100 - . Pri TTL-signalu $0 - 5$ V, val doseže 2.5 V. Tipična vrednost za t_f omenjene povezave na tiskanem vezju je 1 ns na dolžino 15 cm.

Ko pride pri neprilagojeni liniji odbiti val nazaj na začtek linije, se tu ponovno odbije nazaj v linijo, kjer se sreča z drugimi napredujočimi in odbitimi valovi. Lahko si predstavljamo, kakšna zmešnjava signalov je lahko na liniji, če ne posvetimo dovolj pozornosti prilagojenosti linije na njeno breme.

Dobro delujoč prenos signala po kartici ali v čipu zahteva zaključek, terminator. Poznano je šest vrst zaključitve:

- odprta linija (kapacitivno breme, ki ga povzroča vhod naslednje enote), $C_{in} < 20$ pF
- paralelna ohmska zaključitev $R_0 \approx 100$ -
- Théveninova zaključitev, $R_1 \approx 300$ - na V_{DD} in $R_2 \approx 100$ na GRND
- Serijska ohmska zaključitev na vhodu $R_1 \approx 50$ - , če je $Z_0 = 100$ -
- paralelna zaključitev z $R_0 \approx 100$ - s pozitivno prednapetostjo
- paralelna zaključitev z $R_0 \approx 100$ - s serijskim kondenzarorjem $C_1 \approx 100$ pF.

Osnovni namen, ki ga imamo pri prenašanju signalov, je, da prenesemo čim več energije od izvora do ponora signalov. Pri tem je potrebna električna napetost, električni tok in, da pridemo do energije, tudi čas. Največjo energijo prenesemo, če je sistem prenosa uravnotežen, to je v primeru kritičnega dušenja na liniji, ko je $R_S = Z_0 = R_L$. V tem primeru ni nobenih odbojev, polovica energije se potroši na izvornem uporu R_S in polovica na bremenskem uporu R_L. Če imamo $R_L > Z_0$, se del energije odbije in vrača nazaj na vhod, kar je za RSS škodljivo. V kolikor bi vzeli $R_L < Z_0$, bi prišlo na izhodu do večje potrošnje energije, kot jo je na voljo. Ker to ni mogoče, pride do ustreznih odbojev. V splošnem pride lahko pri nepravilno zaključenih linijah do tako velikih povratnih vplivov, da so možne celo okvare in ne samo nepravilno delovanje. Iz izkušenj izhaja, da je za dobro delovanje RSS potrebno vzeti R_L, ki je nekoliko manšji od Z_0.

9.3 METASTABILNOST

Če pogledamo programirne PLD ali GA, vidimo, da imajo te strukture makrocelice tudi registrskega tipa ali decizijskega tipa na osnovi registrskega

tipa makrocelice. Pri takšnih makrocelicah obvezno sodelujejo pomnilne celice ali pomnilni zatiči. Ker gre za programirne strukture, je načrtovalcu dopuščeno, da dela tako s sinhronskimi kot tudi asinhronski signali. Dogodi se, da asinhronski signali toliko nepredvidljivi (nepreučeni s strani načrtovalca), da se sinhronska celica zaradi njenega tehnološkega ozadja ne more odločiti za dvojiško stanje (upset situation). Če nastopi asinhronski signal preblizu ali v ozkem oknu, v katerem se sinhronska celica odloča, upoštevajoč urni signal, za stanje 0 ali 1, pride do stanja, ki ni niti 0 in niti 1 (metastanje). Pri samo sinhronskih vhodih ni problemov, ker je z razdaljo med dvema urnima impulzoma vse determinirano že s projektom čipa, iz katerega je razvidno, pri kakšni hitrosti ure sistem deluje povsem stabilno. Verjetnost takšne metastabilnosti je sicer majhna, kritično pa je to, da je večja od 0. Verjetnost metastablnosti je določena eksperimentalno z enačbo

$$p = T_0 \exp \frac{-t_r}{\tau_c} \tag{9.5}$$

Pri tem sta časa T_0 in τ_c konstanti, ki izhajata iz osnovnega načrtovanja vezja in t_r resolucijski čas (čas da pomnilna celica pride iz nestabilnega v stabilno stanje). Vrednosti iz prakse so na primer: $t_r = 5$ ns, $\tau_c = 0.1$ ns ter $T_0 = 0.1$ ns, kar da verjetnost $p = 2\ 10^{-23}$ [15]. Druga možna mera za meta-nestabilnost je MTBU (mean-time between upset, podobno kot MTBF v računalniški zanesljivosti). Ta je

$$\text{MTBU} = \frac{1}{p\, f_c\, f_d}\,. \tag{9.6}$$

Verjetnost p že poznamo, f_c je frekvenca ure in f_d frekvenca podatkov.

Zgled 10 *:*

Pri podatkih, ki smo jih navedli pri gornjem izračunu za p in podatkih $f_c = 133$ MHz ter $f_d = 2$ MHz izračunajmo MTBU*.*

$$\text{MTBU} = \frac{1}{2\ 10^{-23} \cdot 133\ 10^6 \cdot 2\ 10^6} = 19\ 10^7 \text{ s}.$$

Zadnje pomeni, da v poprečju približno 6 let lahko pričakujemo pomoto zaradi metastabilnosti.

9.4 PRESLUH

Mnogo vzrokov zahteva, da so RSS majhne v prostorskem pogledu. Volumen RSS, prenosnih linij, čipov, pomnilnikov itd. se hitro manjša, hrati pa se hitro povečuje hitrost njihovega delovanja. Pri današnjih hitrostih delovanja in paketiranosti komponent obstaja vrsta medsebojnih vplivov, med katere sodi tudi presluh. Če sta na primer sosednja priključka na čipu blizu skupaj, obstaja znatna medsebojna (motilna) kapacitivnost, ki povzroča, da spremembo signala ne čuti samo pravi priključek, temveč tudi sosednji priključek. Na sosednjem priključku se pojavijo motilni signali. Če imamo na tiskanini ali na GA dve dolgi vodili, ki potekata več ali manj paralelno, pride do motenj elektromagnetnega oziroma induktivnega značaja. Opisane in druge motnje zmanjšujemo z vzemljitvenimi in drugimi pomožnimi plastmi na tiskanini ali drugačni podlagi. Omenjene in podobne motnje induktivnega ali kapacitivnega izvora med motilcem in motencem v RSS imenujemo presluh (crosstalk).

Poglejmo si najprej kapacitivni presluh med sosednima priključkoma na opazovanem čipu. Priključek z imenom 1 naj ima izhodni signal in priključek z imenom 2 vhodni signal (priključka 1 in 2 sta lahko na katerikoli poziciji čipnih priključkov). Oceno presluha predstavlja razmerje med časovno konstanto motenca $R_i C_M$ in časom T_{10-90} motilca (aktivnega signala)

$$C_{rc} = \frac{R_2 C_M}{T_{10-90}}. \tag{9.7}$$

Pri tem je R_2 [-] skupna upornost priključka 2, C_M [pF] kapacitivnost med motilcem 1 in motencem 2. T_{10-90} [s] je čas večinskega poteka signala motilca 1. Naj je na primer na priključku 2 100-ohmska linija, na priključku 1 pa izhod s časom naraščanja $T_{10-90} = 5$ ns ob tem, da med priključkoma 1 in 2 obstaja medsebojna kapacitivnost $C_M = 4$ pF. Presluh med 1 in 2 je tako

$$C_{rc} = \frac{50 \cdot 4\ 10^{-12}}{5\ 10^{-9}} = 40\ 10^{-3} = 4\ \%\,.$$

Upornost priključka 2 je paralelni sestav zaključne upornosti 100 - in karakteristične impedance linije $Z_0 = 100$ - , katero smo priključili na 2.

Omenjeni kapacitivni presluh zmanjšamo pri ASIC-načrtovanju tako, da damo na visoko-ohmski priključek 2, z vhodnim signalom, dodatno kapacitivno breme. Če je na primer na priključku 1 urni signal in na priključku 2 visoko-ohmski vhod S za SR-pomnilno celico, damo na priključek 2 dodatno kondenzator C = 0.01 μF proti GRND. Presluh se zmanjša na razmerje

$$C_{rc} = \frac{C_M}{C} = \frac{4\ 10^{-12}}{0.01\ 10^{-6}} = 0.0004\ \%,$$

kar je zanesljivo zanemarljivo. V kolikor je visoko-ohmska zaključitev priključka 2 enaka R_S =10 K- , imamo časovno konstanto vhoda S, ki je T_S = $C\,R_S$ = 0.1 ms.

Kako pridemo do medsebojne kapacitivnosti C_M? Brez te po izrazu (9.7) ne moremo izračunati opazovanega presluha C_{rc}. Za ustrezno čipno izvedbo moramo poiskati empirični podatek, ki ga najdemo v katalogih uporabljenih IC-vezij. Iz nadalnje razpave o presluhu pa izhaja, da C_M lahko določimo tudi laboratorijsko.

Obravnavani presluh ni edini, s katerim se ubada današnji načrtovalec RSS. Vpliven prisluh imajo tudi komponente, če so te, glede na dinamiko signalov, blizu skupaj in imajo določeno dolžino paralelnosti. Pri tem obstajata medsebojna induktivnost in tudi medsebojno kapacitivnost, ki povzročata tako induktivni C_{rl} kot tudi kapacitivni presluh C_{rc}. Pri hitrih ocenah v laboratoriju lahko celoten presluh izračunamo kar z vsoto C_r = $C_{rl} + C_{rc}$. V takšnem primeru moramo znati sami določiti C_M in L_M. Namreč, ti dve postavki sta produkt našega načrtovanja in za naše (subjektivno) načrtovanje nam drugi ne morejo dati podatkov. Pomagamo si z meritvijo signala, ki ga merimo z osciloskopom na moteni strani, če na motilni strani nastopi stopničasti potek napetosti. V tem primeru imamo [17]

$$\begin{aligned} C_M &= \frac{P}{R_2\ \triangle V} \\ L_M &= \frac{P\ 2R_1}{\triangle V}\,. \end{aligned} \tag{9.8}$$

P [pVs] je površina motilnega signala, če imamo napetostno stopnico za $\triangle V$ [V] na motilni strani. R_2 [-] je upornost motene in R_1 motilne strani. Ustrezna presluha sta

$$C_{rc} = \frac{R_2\ C_M}{T_r}$$
$$C_{rl} = \frac{L_M}{2R_1\ T_r}\,. \tag{9.9}$$

C_M sta merjeni v pF in L_M v nH.

Poglejmo si še presluh, ki nastane med dvema paralelnima vodnikoma (povezavama), ki potekata nad površino GRND oziroma ustrezno podlago (na primer GRND plast tiskanega vezja). Ne gre samo za dve povezavi kar tako, temveč za dve tokovni zanki z ustreznima magnetnima poljema. Induktivni presluh za takšna dva vodnika je

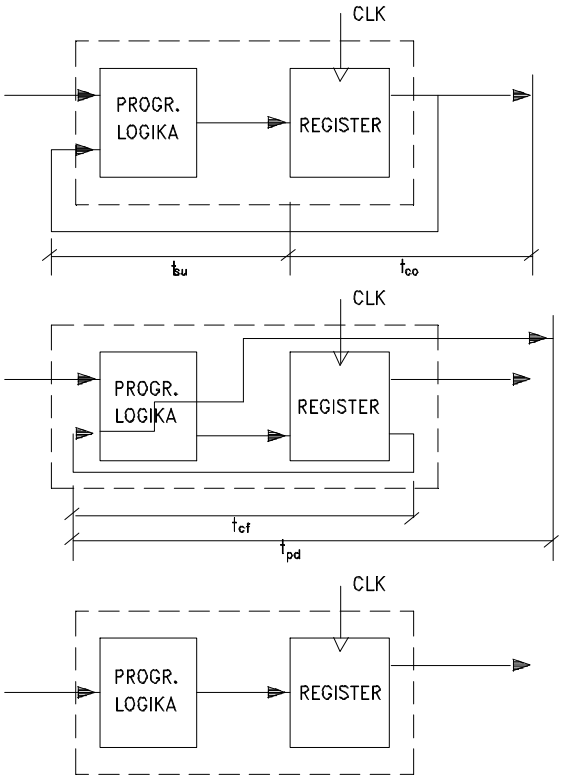
$$C_{rl} \simeq \frac{K}{1 + \left(\frac{d}{h}\right)^2}\,. \tag{9.10}$$

h je višina nad ravnino GRND in d razdalja med vodnikoma. Konstanta K zavisi od časa naraščanja signala na vodnikih in od dolžine paralelnosti teh vodnikov. K je vedno manjša od 1.

V kolikor sta omenjena vodnika dva sosedna priključka na čipu, imamo medsebojne induktivnosti in kapacitivnosti: pri čipu s 14 priključki reda (podnožje DIP) $L_M = 8$ nH in $C_M = 4$ pF ter pri čipu z 68 priključki (podnožje PLCC) $L_M = 77$ nH in $C_M = 7$ pF. Vidimo, da gostota priključkov močno povečuje medsebojne motnje. Globalno analizo najrazličnejših presluhov v RSS podaja [17].

9.5 POVRATNA VEZAVA MAKROCELIC

Pri obravnavi različnih tehnik načrtovanja (PAL, GAL, LCA itd.) smo omenili vhodno-izhodne celice, ki se dodajajo programirnim poljem oziroma matrikam. V celicah imamo tudi pomnilne celice, ki funkcijsko zelo dopolnijo sposobnost tehnike načrtovanja. Na primer OLMC na GAL-čipih lahko vsebujejo pomnilne celice ali pa tudi ne. Če jih vsebujejo, lahko na osnovi njih dobimo ragistrska delovanja, lahko pa iz takšnih celic potegnemo tudi decizijsko delovanje. V omenjenih celicah je prisotna ura, s programirnim poljem pa ustvarjamo tudi asinhronske signale. Med temi asinhroskimi signali in uro mora biti določena usklajenost sicer pride lahko do težav.



Slika 9.2: Različni časi pri različnih načinih delovanja OLMC v GAL-tehniki.

Na sliki 9.2 vidimo tri primere uporabe registrske vhodno-izhodne celice. Zgoraj vidimo zunanjo povratno povezavo, na katero se nanašata časa t_{su} in t_{co}. Prvi pove, koliko časa mora predhodno obstajati veljavni vhodni signal na programirni logiki, preden učinkuje urni impulz na dogajanje. t_{co} je čas, ki je potreben po urnem vplivu, da pridemo do izhodnega signala. Važna je performansa $f_{\max} = 1/(t_{\mathrm{co}} + t_{\mathrm{su}})$, na osnovi katere pridemo od maksimalne možne frekvence ure. Pri čipu GAL16V8B je na primer v najboljšem slučaju $t_{\mathrm{su}} = 2$ ns in $t_{\mathrm{co}} = 10$ ns. Vidimo, da na registrski del vezja odpade precej večja zakasnitev kot na programirno polje. Performansa pri omenjenih vrednostih je $f_{\max} = 41.6$ MHz.

Osrednji del slike 9.2 se nanaša na notranjo povratno povezavo, za

katero velja $f_{\max} = 1/(t_{\mathrm{cf}} + t_{\mathrm{pd}})$. Čas t_{cf} je zakasnitev, ki jo med pojavom vplivnega urnega impulza in registrsko (notranjo) povratno vezavo. Čas t_{pd} predstavlja zakasnitev skozi programirno polje do vhoda-izhoda. Pri omenjenem GAL-čipu je v najboljšem primeru $t_{\mathrm{cf}} = 0$ ns (v najslabšem pa 6 ns) in $t_{\mathrm{pd}} = 3$ ns. Performansa znaša $f_{\max} = 45.4$ MHz.

Poglejmo si še spodnji primer na sliki 9.2. Vidimo, da nimamo nobene povratne povezave, zato nimamo nobenih težav. V tem primeru je ura lahko najhitreša in pri opazovanem čipu znaša $f_{\max} = 50$ MHz.

Primer nazorno kaže, na kakšen način moramo spremljati časovnost, ko gradimo logične sheme na osnovi programirnih polj in vhodno-izhodnih celic.

9.6 URA

Pri specifikacijah RSS se je v preteklosti dogajalo to, da so RSS opisovali posebej in urni sistem posebej. Največkrat je računalnikar zelo malo vedel o ozadju urnih signalov, jemal jih je kot nekaj idealnega, logičnega. Z desetkratnim povečanjem hitrosti urnega sistema v dobi petih let se je tudi pri urnem sistemu marsikaj spremenilo. Načrtovalec programirne RSS mora skrbeti celo za usklajenost med urnim sistemom in asinhronskimi signali sicer lahko RSS pade v metastabilnost ali v druge težave.

Vedno bolj je pomemben najslabši čas kot posledica nesimetričnosti vpliva urnega signala. Kaj to pomeni, si poglejmo na primeru dveh D-pomnilnih celic, ki sta med seboj serijsko vezani in napajani preko različno dolgih urnih vodil. Med izhod Q_1 in vhod Q_2 vstavimo vrata G. Najslabši čas (worst case), da pride signal preko vrat G, je

$$T_{slow} = T_{C1,\max} + T_{FF,\max} + T_{G,\max} \tag{9.11}$$

Pri tem je $T_{C1,\max}$ največja zakasnitev povezave med urnim izvorom in urnim vhodom na prvi celici, $T_{FF,\max}$ je največja zakasnitev, da ura vpliva na izhod Q_1 ter $T_{G,\max}$ največja zakasnitev, da pride signal skozi vrata G (vključno s segmenti povezav na vhodu in izhodu).

Signal iz vrat G je vezan na vhod D_2 in je pogojen z uro v drugi D-pomnilni celici. Naj se ustrezni urni signal pojavi v času T_{CLK}. Ta je zakasnjen za minimalni čas $T_{C2,\min}$ (zakasnitev povezave urnega izvora do urnega priključka na drugi celici), preden pride do urnega vhoda druge celice, CLK2. Druga celica zahteva veljaven vhod najmanj T_{setup} pred

signalom na CLK2. Vstopni čas, ki ga zahteva pravilno delovanje druge celice, je

$$T_{req} = T_{CLK} + T_{C2,\min} - T_{setup}\,. \tag{9.12}$$

Signal iz vrat G mora priti pred T_{req}, da lahko pravilno postavi drugo celico. To pomeni, da je $T_{slow} < T_{req}$ oziroma

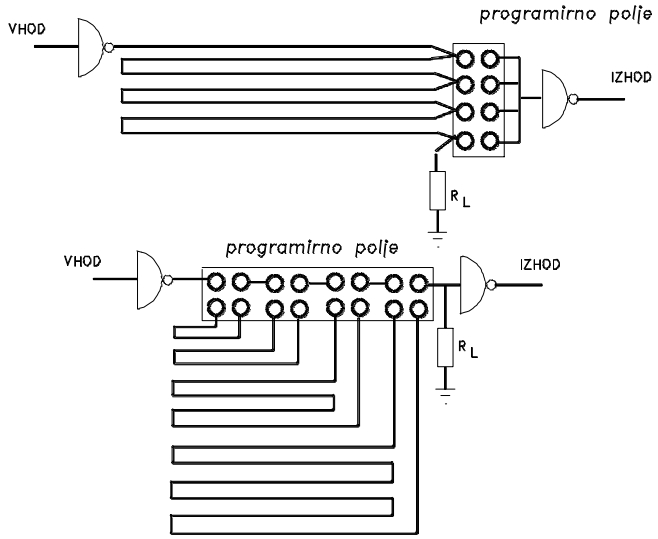
$$T_{CLK} > T_{FF,\max} + T_{G,\max} + T_{setup} + (T_{C1,\max} - T_{C2,\min})\,. \tag{9.13}$$

T_{CKL} je pravzaprav urna perioda. Če je ta prekratka, frekvenca ure previsoka, pride do nezadovoljivega delovanja opazovanega sinhronskega preklopnega vezja. Primer nazorno kaže, da moramo pri logičnem načrtovanju vseskozi paziti, kaj je s časovnimi odnosi v opazovanem detajlu načrtovanja.

Da zmanjšamo razlike zakasnitev urnih signalov, ki vodijo do različnih delov logične sheme, obstajajo različni koncepti polaganja urnih vodil. Tako imamo:

- Steienerjevo drevo (glej razdelek 2.6.1) s centralno točko, ki postavlja enako dolge segmente vodil. Močnostni gonilnik ure je tik pred Steinerjevo točko.
- Drevesna struktura urnih vodil, kjer vsaka veja poganja enako število vrat, kar naj bi zmanjševalo nesimetričnosti ure.
- Urna linija, ki je najenostavnejši in zato najbolj pogost primer vodil. Zavedati se moramo, da vsak odjem urnega signala vzdolž linije povečuje čas dvigovanja urnega signala. Pa ne samo to, vračajo se nazaj tudi odbiti signali, katerih velikost dosega $-C(Z_0/2)$.

Pomembna je pravilna zaključitev urne linije. Uveljavljati so se začele tudi programirne dolžine urnega vodnika. Med točkami, ki so blizu skupaj, imamo faktorsko povečevanje vodnika, tako da imamo lahko praktično v isti točki dolžino L ali $2L$ ali $3L$ itd. Izberemo (programiramo tisto, ki na opazovanem mestu načrtovanja najbolj ustreza; programiranje je lahko izvedeno z jahači). Na sliki 9.3 vidimo dva primera programirnega prilagajanja vodil [17]. V prvem primeru se povečuje dolžina z eno enoto, medtem ko imamo v drugem primeru povečevanje večje: $T, 2T, 4T, 6T$ itd. Ni nujno, da ta princip prilagojevanja zakasnitev uporabljamo samo v primeru



Slika 9.3: Programirna vodila urnih signalov.

urnega sistema. Uporaben je povsod tam, kjer imamo velike tokovne pretoke in je ob tem potrebno prilagojevanje zakasnitev. Sicer pa obstajajo na tržišču splošni zakasnilni elementi s specificiranimi zakasnitvami.

Programirno polje na sliki 9.3 je tako narisano, da za programiranje potrebujemo jahače. V gornjem primeru urne linije potrebujemo samo en jahač, katerega postavimo na želeno mesto. V spodnjem primeru potrebujemo zatiče za eno vrsto (4). Zataknjeni so vodoravno v zgornji vrstici. Ko izberemo dolžino, na ustreznem mestu postavimo dva zatiča navpično, ostale pa pustimo vodoravno.

Del IV

POSTOPKI NAČRTOVANJA RSS

Poglavje 10

PROGRAMIRNO NAČRTOVANJE AVTOMATOV

10.1 UVOD

V času klasičnega računalništva smo pri načrtovanju avtomatov uporabljali definicije Mooreovega in Mealyjevega avtomata ter teorijo regularnih izrazov. Na osnovi tega smo dobili dokaj optimalne (minimalne) rešitve, ki imajo za današnji čas to slabost, da ne upoštevajo dovolj modularnosti in že napol vnaprej pripravljene programirne gradnike (PAL, GAL, PLD, FPGA itd.), ki se dobijo na tržišču. Današnje rešitve ne težijo v detajlnje optimizacije in minimizacije, ker za to ni nobene potrebe, če le pritegnemo v realizacije omenjene programirne gradnike. S tem pa nismo rekli, da pri današnjih realizacijah avtomatov ni potrebna nobena optimizacija. V tem poglavju se bomo bolj pogovarjali o možnih ralizacijah, kot o osnovni teoriji avtomatov.

10.2 MATRIČNO GLEDANJE NA PRAVILNOSTNO TABELO

Slika 3.23 prikazuje sestavo dekodirne in kodirne matrike v smislu bralnega pomnilnika oziroma realizacije logične funkcije. Matriki sta obe programirni, ali ena od obeh, ali nobena od obeh. Tudi če sta obe matriki programirni, dekodirna ni demultipleksorska in kodirna ni multipleksorska.

x	**y**
D	K

Tabela 10.1: Pravilnostna tabele matrik logične funkcije.

Kodirna matrika bi na primer postala multipleksorska, če bi bili elementi matrike spremenljivi s procesorsko hitrostjo. Programirna kodirna matrika res dopušča spreminjanje elementov, vendar ne s procesorsko hitrostjo. Sodeč po povedanemu, programirna kodirna matrika tudi ni pravi kodirni gradnik, saj z določeno majhno hitrostjo vendarle dopušča spremembo elementov matrike. Podobno imamo relacije tudi pri dekodirni oziroma demultipleksorski matriki.

Vzemimo pravilnostno tabelo 10.1. Ta predstavlja izhodni vektor $\mathbf{y} = (f_1, f_2, ..., f_m)$ v odvisnosti od vhodnega vektorja $\mathbf{x} = (x_1, x_2, ...x_n)$. Pri tem so elementi $f_i = f_i(x_1, x_2, ..., x_n)$ običajne logične funkcije ali pa tudi časovne, če sta vektorja $\mathbf{x}$ in $\mathbf{y}$ časovno premaknjena, $\mathbf{y} = \mathrm{D}^k\mathbf{x}$. Pri tem je k lahko pozitivno, negativno, ali število 0. V prvem primeru gre za bodočnost, v drugem za preteklost in v zadnjem, za sedanjost [1]. D je (programirna ali neprogramirna) dekodirna matrika in K (programirna ali neprogramirna) kodirna matrika.

Matrika 10.1 nam dovoljuje, da zapišemo vse logične funkcije f_i v PDNO na osnovi izraza [1]

$$\mathbf{y} = (\mathbf{x}\,\& \equiv D') \vee \&K\,. \tag{10.1}$$

Del izraza v oklepaju nam predstavlja naslove, medtem ko nam nadaljnji del izraza predstavlja branje podatkov - pri prispodobi, da beremo iz pomnilnika funkcijske vrednosti kot podatke, ki so shranjeni pod omenjenimi naslovi. Izraz v oklepaju je mintermski vektor $\mathbf{m} = \mathbf{x}\,\& \equiv D'$, ki ima dolžino 2^n, $\mathbf{m} = (m_0, m_1, ..., m_{2^n-1})$. Vedno bomo imeli minterm m_i za naslov (celice v pomnilniku). Podobno kot smo ravnokar pristopili k PDNO logične funkcije f_i, lahko pristopimo k PKNO iste f_i, $i = 1, 2, ...m$. Za PKNO velja

$$\mathbf{y} = (\mathbf{x} \vee \equiv D')\& \vee K\,. \tag{10.2}$$

V tem primeru naslavljamo z makstermi $\mathbf{M} = (M_{2^n-1}, M_{2^n-2}, ..., M_0)$. V primeru m_i se naslavljanje celice i opravlja z vrednostjo 1 (vse ostale celice imajo vrednost naslavljanja 0) in v primeru M_{2^n-1-i} se naslavljanje celice

$2^n - 1 - i$ opravlja z vrednostjo 0 (vse ostale celice imajo vrednost naslavljanja 1), $i = 1, 2, ...m$.

V izrazih (10.1) in (10.2) imamo med dvema spremenljivkama dve operaciji, kar v matričnem računu ni v navadi. Omenjena, Iversonova, notacija izhaja iz APL-jezika [1], v katerem je matrika A označena z $A^{1:a}_{1:b}$, kjer je a število vrstic in b število stolpcev. Dvojna matrična operacija je tako v splošnem podana

$$\begin{aligned} C^{1:n}_{1:m} &= A^{1:n}_{1:t} \circ \bullet B^{1:t}_{1:m} \\ c^i_j &= \circ/(a^i_{1:t} \bullet b^{1:t}_j)\,. \end{aligned} \tag{10.3}$$

$\circ$ in $\bullet$ sta poljubni (navadni ali logični operaciji), / pa je operacija redukcije. Ta operacija nam po določeni operaciji reducira vektor na skalar, matriko na vektor itd. Glede na običajno notacijo v matričnem računu je $c^i_j = c_{ij}$.

Zgled 11 *:*

Določimo vektor $\mathbf{y}$ *za logični funkciji v pravilnostni tabeli*

x_1	x_2	f_1	f_2
0	0	1	1
0	1	0	0
1	0	1	0
1	1	1	1

.

Iz dane tabele dobimo

$$\begin{aligned} \mathbf{x} &= [x_1, x_2] \\ \mathbf{y} &= [f_1, f_2] \\ D' &= \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \\ K &= \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}. \end{aligned}$$

Po izrazu (10.3) *si najprej izračunajmo mintermski vektor*

$$\begin{aligned}
\mathbf{m} &= \mathbf{x}\,\& \equiv D' = [x_1, x_2]\,\& \equiv \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \\
&= \&/[(x_1 \equiv 0), (x_2 \equiv 0); (x_1 \equiv 0), (x_2 \equiv 1)\,; (x_1 \equiv \\
&\equiv 1), (x_2 \equiv 0); (x_1 \equiv 1), (x_2 \equiv 1)] \\
&= \&/[\bar{x}_1, \bar{x}_2; \bar{x}_1, x_2; x_1, \bar{x}_2; x_1, x_2] = \\
&= [\bar{x}_1\bar{x}_2, \bar{x}_1 x_2, x_1\bar{x}_2, x_1 x_2] = [m_0, m_1, m_2, m_3].
\end{aligned}$$

Sedaj ko imamo vektor naslavljanja $\mathbf{m}$*, lahko izračunamo izhodni funkciji*

$$\begin{aligned}
\mathbf{y} &= \mathbf{m} \vee \&K = [m_0, m_1, m_2, m_3] \vee \& \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \\
&= \vee/[(m_0 1, m_1 0, m_2 1, m_3 1); (m_0 1, m_1 0, m_2 0, m_3 1)] = \\
&= [m_0 \vee m_2 \vee m_3,\ m_0 \vee m_3] = [x_2 \rightarrow x_1,\ x_1 \equiv x_2]\,.
\end{aligned}$$

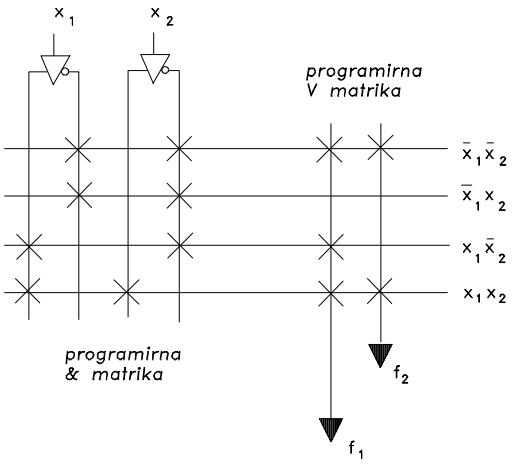
f_1 je implikacija in f_2 ekvivalenca spremenljivk x_1 in x_2.

Noben PAL- ali GAL-gradnik ni tako majhen, da bi gornji funkciji v večji meri pokrili njegovo programirno polje in število priključkov. Pa vendar, na sliki 10.1 *vidimo FPAL princip programiranja za dani zgled.*

10.3 MATRIČNA POSTAVITEV AVTOMATA

Iz teorije končnih avtomatov izhaja, da je končni avtomat podan kot petorka $A=\{X, Y, Z, \lambda, \delta)$ [1]. V tej petorki je X vhodna abeceda, Y notranja abeceda in Z izhodna abeceda. V abecedah so črke v smislu množic $X=\{x_1, x_2, ..., x_n\}$, $Y=\{y_1, y_2, ..., y_r\}$ in $Z=\{z_1, z_2, ..., z_m\}$. x_i je vhodna črka, y_j je notranje stanje (notranja črka) in z_k je izhodna črka. Med abecadami X, Y in Z obstajajo preslikave

$$\begin{aligned}
&\mathrm{D}^1 y = \delta(x, y) \\
&z = \lambda(x, y)\,.
\end{aligned} \tag{10.4}$$



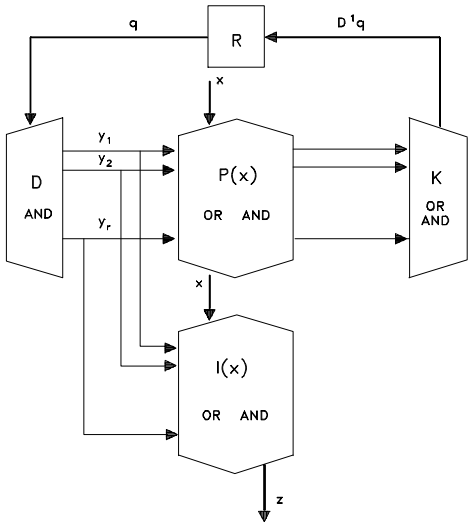
Slika 10.1: *FPAL-predstavitev vektorja* $\mathbf{y}$ *v Zgledu* 11.

x, y in z so črke v sedanjosti, medtem ko je $\mathrm{D}^1 y$ naslednje stanje. Na osnovi naslednjega stanja gradimo časovne sekvence (dogodke), katerim sledi delovanje avtomata A.

Do vhodnih črk lahko pridemo, če imamo na avtomatu $a \geq log_2\, n$ vhodnih dvojiških kanalov v. Podobno je potrebno imeti na izhodu $c \geq log_2\, m$ kanalov f. Ker je δ časovna funkcija, za stanja avtomata A potrebujemo pomnilnik. r stanj nam lahko da $b \geq log_2\, r$ dvojiških pomnilnih celic z izhodi q, ki sestavljajo omenjeni pomnilnik. Pri programirnem postavljanju avtomata, ko uporabljamo programirne gradnike, pomnilnik enostavno imenujemo kar register.

Model avtomata, ki je primeren pri načrtovanju avtomatov na osnovi matrik, vidimo na sliki 10.2 (za gradnike modela glej [18], [1]. R je register s b celicami. Ta vpliva na matriko D z vektorjem $\mathbf{q} = (q_1, q_2, ..., q_b)$, kar oblikuje stanja $y_i, i = 1, 2, ..., r$. Ta stanja vplivajo na matriko prehajanja P in izhodno matriko I. Iz matrike P dobimo na osnovi matrike K krmilne funkcije celic v registru R. Vhodni vektor $\mathbf{x} = (x_1, x_2, ..., x_a)$ vnaša svoj vpliv v matriki P in matriki I (zadnje le, če je to potrebno). Vse izhodne črke nam da vektor izhodnih kanalov $\mathbf{f} = (f_1, f_2, ..., f_c)$.

Vsaka od matrik na sliki 10.2 je lahko programirna ali pa tudi ne. V registru so v primeru PLD-gradnikov največkrat celice D-tipa, možne pa so tudi druge z vhodi RS, JK itd. Slika nazorno prikazuje, zakaj imajo



Slika 10.2: Matrični model končnega avtomata.

PLD-gradniki poleg decizijskih programirnih polj tudi pomnilne celice.

Sistem enačb (10.4) velja za Mealyjev avtomat. Njegova značilnost je, da ima dinamične izhodne črke. Izhodno črko dobimo samo v času spremembe staja $y \rightarrow \mathrm{D}^1 y$ (zelo kratek čas) v odvisnosti od vhodne črke x. Če hočemo dobiti staconarne izhodne črke, nam je na voljo Mooreov avtomat. V njem je določena izhodna črka z prisotna toliko časa, dokler se avtomat nahaja v določenem stanju y, pri tem pa vhodna črka ne igra nobene vloge. Za Mooreov avtomat A imamo tako

$$\begin{aligned} \mathrm{D}^1 y &= \delta(x, y) \\ z &= \lambda(y)\,. \end{aligned} \tag{10.5}$$

Model na sliki 10.2 lahko opišemo z Iversonovo notacijo

$$\begin{aligned} \mathrm{D}^1 \mathbf{y} &= \mathbf{y} \vee \&P(\mathbf{x}) \\ \mathbf{z} &= \mathbf{y} \vee I(\mathbf{x})\,. \end{aligned} \tag{10.6}$$

Povezava med vektorjem $\mathbf{y}$ in $\mathbf{q}$ je naslednja

$$\mathbf{y} = \mathbf{q}\& \equiv D' \,. \tag{10.7}$$

D' je transponirana dekodirna matrika. Na osnovi tega izraza lahko izraza (10.6) zapišemo z vektorjem $\mathbf{q}$

$$\begin{aligned} &\mathrm{D}^1\mathbf{q} = (\mathbf{q}\& \equiv D') \vee \&P(\mathbf{x}) \vee \&D \\ &\mathbf{z} = (\mathbf{q}\& \equiv D') \vee \&I(\mathbf{x})\,. \end{aligned} \tag{10.8}$$

Matrika prehajanja stanj $P = [p_{st}]$ ima red $r \times r$, p_{st} je vhodna črka (disjunkcija vhodnih črk) x, ki nas vodi iz sedanjega stanja y_s v naslednje stanje y_t. Če ni prehoda, je ustrezen element $p_{st} = 0$. Izhodna matrika Mooreovega avtomata je $I = [w_{st}]$ in ima red $r \times m$. Velja $w_{st} = 1$, če je pri stanju y_s izhodna črka z_t sicer je $w_{st} = 0$. Pri Mealyjevem avtomatu je izhodna matrika odvisna tudi od vhodne črke x. Imamo $w_{st} = x$, ko prehod iz stanja y_s zahteva vhodna črka x in pri tem zahteva izhodno črko z_t. Če prehod iz stanja y_s pri x in z_t ne obstaja, je $w_{st} = 0$.

V splošnem modelu avtomata smo stanje označili z y. V kolikor gre za Mooreov avtomat označujemo stanja z b in v primeru Mealyjevega avtomata z a, kar nam takoj pove, s kakšnim avtomatom imamo opraviti.

Matematični model (10.8) podaja delovanje končnega avtomata, če so v registru uporabljene D-pomnilne celice. Pri teh celicah velja, da je $d = \mathrm{D}^1 q$, kar pomeni, da je vhod celice d enak izhodni funkciji celice. Torej nam krmilnih funkcij v pravem pomenu besede sploh ni potrebno računati. V PPAL in drugih gradnikih imamo sicer največkrat (v registrskih celicah in načinih delovanja) D-celice. Vedno pa ni tako. Vzemimo, da imajo gradniki, s katerimi realiziramo avtomat, JK-celice. V tem primeru je potrebno matriko K na sliki 10.2 zamenjati z matriko K^*. Takšno zamenjavo si poglejmo kar na primeru.

Vzemimo, da je naš detajl načrtovanja podan s tabelo 10.2. $d_1 = \mathrm{D}^1 q_1$ in $d_2 = \mathrm{D}^1 q_2$ sta krmilni funkcij pomnilnih celic D_1 in D_2. Iz vhodov d_1 in d_2 moramo preiti na vhode j_1, k_1, j_2, k_2, ker ima programirni gradnik, ki naj pokrije tabelo 10.2, JK-pomnilne celice. Stolpci j_1, k_1, j_2, k_2 v dani tabeli so določljivi na osnovi pravilnostne tabele za JK-celico [1] oziroma izhodne funkcije

$$\mathrm{D}^1 q = j\bar{q} \vee \bar{k}q.$$

x_1	x_2	q_1	q_2	D^1q_1	D^1q_2	j_1	k_1	j_2	k_2
0	1	0	0	1	0	1	b_0	0	d_0
0	1	0	1	–	–	–	–	–	–
0	1	1	0	0	1	a_2	1	1	d_2
0	1	1	1	x	x	x	x	x	x
1	0	0	0	0	1	0	b_4	1	d_4
1	0	0	1	0	1	0	1	c_5	0
1	0	1	0	0	0	1	b_6	0	d_6
1	0	1	1	x	x	x	x	x	x

Tabela 10.2: Pravilnostna tabela sekvenène strukture.

Konstante a_i, b_i, c_i in d_i so poljubne Booleove konstante. Izberemo jih tako, da so krmilne funkcije čim bolj enostavne.

Če sta izhoda matrike K vhoda d_1 in d_2 za register $\mathsf{R_D}$, so izhodi matrike K^* vhodi j_1, k_1, j_2, in k_2 za register $\mathsf{R_{JK}}$.

10.4 PROGRAMIRNOST AVTOMATA

10.4.1 Programiranje Mooreovega avtomata

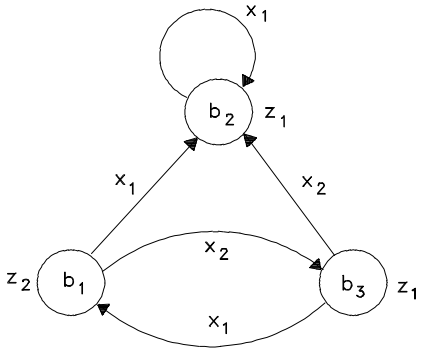
V prejšnjem razdelku smo spoznali model Mooreovega avtomata, na osnovi katerega lahko pristopimo k ročni ali programirni oziroma avtomatizirani realizaciji. Zanima nas predvsem programirna realizacija, saj imamo, kot načrtovalec, v tem primeru najmanj dela, seveda pa moramo imeti na voljo ustrezno računalniško orodje. Računalniško orodje ne izbiramo glede na vrsto, tip itd. avtomata, temveč glede na gradnike, s katerimi želimo avtomat realizirati. Tako nam v primeru PAL- ali PPAL-gradnikov pride prav ABEL, v primeru ALTERA-gradnikov A+PLUS, v primeru Xilinx-gradnikov XACT itd.

Oglejmo si, kako bi Mooreov avtomat, ki je podan na sliki 10.3, realizirali s PPAL-gradniki uporabljajoč orodje ABEL. Na omenjeni sliki je avtomat podan v obči obliki (na abstraktnem nivoju), ki jo pa moramo približati realizaciji. Približanje opravimo s konkretnim kodiranjem vhodnih, notranjih in izhodnih črk. Vzemimo naslednji kodni tabeli

b	q_1	q_2
b_1	0	0
b_2	0	1
b_3	1	0

z	**f**
z_1	0
z_2	1

.



Slika 10.3: Diagram prehajanja stanj Mooreovega avtomata.

Pri teh kodah imamo naslednji ABEL-izvorni program (ABEL-HDL Module)

```
MODULE moore
    x1 pin 2;
    x2 pin 3;
    q1 pin 12 ISTYPE 'reg';
    q2 pin 13 ISTYPE 'reg';
    f pin 14 ISTYPE 'com';
    b1 = !q1 * !q2;
    b2 = !q1 * q2;
    b3 = q1 * !q2;
    nb1 = b3 * x1;
    nb2 = b1 * x1 # b2 * x1 # b3 * x2;
    nb3 = b1 * x2;
    z1 = b2 # b3;
    z2 = b1;
EQUATIONS
    q1 := nb3;
    q2 := nb2;
    f = z2;
END
```

Ker je avtomat zelo majhen, z orodjem ABEL predhodno izberemo najmanjši PPAL-gradnik, ki lahko pokrije diagram prehajanja na sliki 10.3

(če bi izbrali večji gradnik, bi prišli še do večje redundance, kot jo vidimo v poročilu o realizaciji, ki sledi, page 4). Ta je P16LV8R. Vidimo, da smo vhodna kanala x_1 in x_2 dali na priključka 2 in 3. Izhodna kanala q_1 in q_2 smo dali ustrezno na izhodna priključka 12 in 13 s tem, da smo deklarirali tudi registrsko delovanje ustreznih makrocelic. Izhodna funkcija avtomata je $f = z_2$, $\bar{f} = z_1$. Ta je na priključku 14 s tem, da je način delovanja ustrezne celice decizijski (com, combinational). Pravzaprav je edina prava kodirna tabela potrebna le za notranja stanja, saj sta vhodna in izhodna abeceda dosegljivi z enim dvojiškim kanalom. Vse enačbe za vmesne spremenljivke navajamo pod deklaracijo. Pod `equatios` damo samo tiste enačbe, ki glasijo na izhodne spremenljivke. V našem primeru je to le funkcija f, specificiramo pa kot izhodni spremenljivki tudi q_1 in q_2. Izraze lahko na logičen način odčitamo kar iz diagrama prehajanja stanj. `bi` je sedanje stanje in `nbi` naslednje stanje.

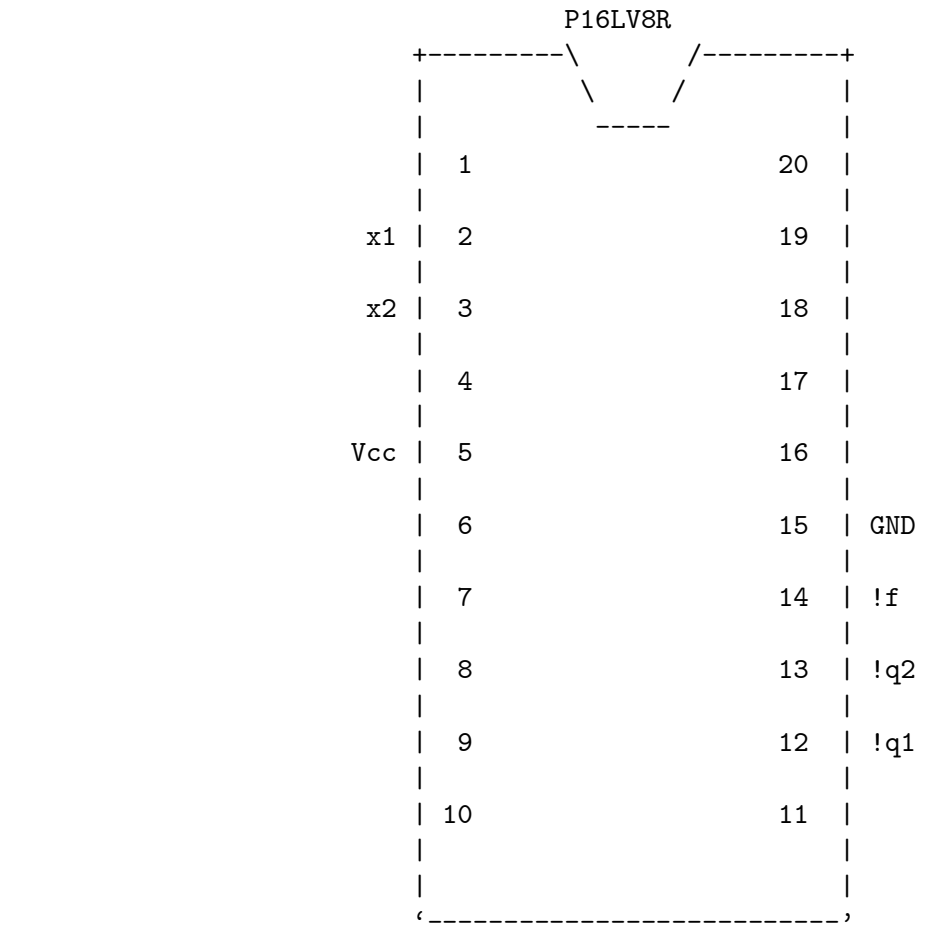
Po procesiranju programa `moore.abl` dobimo poročilo `moore.rep` na šestih straneh. Na vsaki strani je tabela z določenimi podatki. Na strani 4 imamo na primer podatek o redundančnosti čipa, v katerega je ABEL vstavil avtomat. Na strani 3 imamo na preprost negrafičen način podane označitve in pozicije priključkov čipa itd.

```
                                                          Page 1
ABEL 6.10 - Device Utilization Chart  Thu Apr  9 08:19:56 1998
------------------------------------------------------------
Module                   : 'moore'
------------------------------------------------------------
Input files:
    ABEL PLA file        : moore.tt3
    Device library       : P16LV8R.dev
Output files:
    Report file          : moore.rep
    Programmer load file : moore.jed
------------------------------------------------------------


                                                          Page 2
ABEL 6.10 - Device Utilization Chart  Thu Apr  9 08:19:56 1998

P16LV8R Programmed Logic:
------------------------------------------------------------
f     = (  !q1 & !q2 );
q1.D   = !(  !q1 & !q2 & x2 ); '' ISTYPE 'INVERT'
```

```
q2.D   = !(  q1 & !q2 & x2 #   !q1 & x1 ); '' ISTYPE 'INVERT'
```



```
ABEL 6.10 - Device Utilization Chart  Thu Apr  9 08:19:57 1998
P16LV8R Chip Diagram:
---------------------------------------------------------------
```




```
SIGNATURE: N/A
```



```
ABEL 6.10 - Device Utilization Chart Thu Apr  9 08:19:57 1998
P16LV8R Resource Allocations:
---------------------------------------------------------------
        Device        | Resource  |   Design    |
```

```
       Resources      | Available | Requirement | Unused
======================|===========|=============|============
Input Pins:           |           |             |
          Input:      |      9    |      2      |   7 ( 77 %)
Output Pins:          |           |             |
          In/Out:     |      8    |      3      |   5 ( 62 %)
          Output:     |      -    |      -      |   -
Buried Nodes:         |           |             |
          Input Reg:  |      -    |      -      |   -
          Pin Reg:    |      8    |      2      |   6 ( 75 %)
          Buried Reg: |      -    |      -      |   -
```

```
                                                          Page 5
ABEL 6.10 - Device Utilization Chart  Thu Apr  9 08:19:57 1998
P16LV8R Product Terms Distribution:
------------------------------------------------------------
            Signal          |    Pin    | Terms | Terms | Terms
             Name           | Assigned  | Used  |  Max  | Unused
============================|===========|=======|=======|=======
f                           |   14      |   1   |   7   |   6
q1.REG                      |   12      |   1   |   8   |   7
q2.REG                      |   13      |   2   |   8   |   6

     ==== List of Inputs/Feedbacks ====
Signal Name                               | Pin      | Pin Type
==========================================|==========|=========
q1                                        |   12     | BIDIR
q2                                        |   13     | BIDIR
x2                                        |    3     | INPUT
x1                                        |    2     | INPUT
```

```
                                                          Page 6
ABEL 6.10 - Device Utilization Chart Thu Apr  9 08:19:57 1998
P16LV8R Unused Resources:
------------------------------------------------------------
 Pin   |  Pin   |   Product   | Flip-flop
Number |  Type  |   Terms     |   Type
=======|========|=============|==========
    6  |  INPUT |      -      |    -
```

```
 7  |  INPUT |      -      |    -
 8  |  INPUT |      -      |    -
 9  |  INPUT |      -      |    -
10  |  INPUT |      -      |    -
16  |  BIDIR | NORMAL  7   |    D
17  |  BIDIR | NORMAL  7   |    D
18  |  BIDIR | NORMAL  7   |    D
19  |  BIDIR | NORMAL  7   |    D
20  |  BIDIR | NORMAL  7   |    D  .
```

V gornjem poročilu ni programirnih podatkov v JEDEC-formatu. Na strani 1 piše, da so ti podatki v datoteki `moore.jed`, ki zgleda takole:

```
 ABEL 6.10 Data I/O Corp.
JEDEC file for:  P16LV8R V1.3
Created on:  Thu Apr 9 08:19:57 1998
*
QP20* QF2194* QV0* F0*
 X0*
NOTE Table of pin names and numbers*
NOTE PINS x1:2 x2:3 q1:12 q2:13 f:14*
L1280 11111111111111111111111111111111*
L1312 11111111111111111111111101110*
L1536 11110111111111111111111101101*
L1568 01111111111111111111111111110*
L1792 11110111111111111111111101110*
L2048 00000111*
L2120 00000100*
L2128 11111111111111111111111111111111
 11111111111111111111111111111111*
L2193 1*
C1AED*
 70C3
```

Obrazložitev JEDEC-sintakse smo napravili že v 4. poglavju v okviru programiranja PAL-čipov. Postopek programiranja in realizacija le-tega s programatorjem na izhodu računalnika se pri PPAL čipov ne razlikuje od tega, kar smo že povedali v 4. poglavju.

V gornjem primeru smo pri realizaciji avtomata s PPAL-čipom uporabili logične funkcije δ in λ. Orodje ABEL pa sprejme vhodne podatke tudi v

obliki diagrama prehajanja stanj. V zadnjem primeru imamo namesto segmenta `equations` (izhodne funkcije) segment izvornega programa `state_diagram sreg`. Za avtomat na sliki 10.3 imamo tako izvorni program `moore1.abl`

```
MODULE moore1
    x1 pin 2;
    x2 pin 3;
    q1 pin 12 ISTYPE 'reg';
    q2 pin 13 ISTYPE 'reg';
    f pin 14 ISTYPE 'com';
    sreg = [q1, q2];
    b1 = 0; b2 = 1; b3 = 2;
    z1 = 0; z2 = 1;
state_diagram sreg;
 State b1:
    f = z2;
    IF (x1) THEN b2;
    ELSE IF (x2) THEN b3;
 State b2:
     f = z1;
    IF (x1) THEN b2;
 State b3:
    f = z1;
    IF (x1) THEN b1;
    ELSE IF (x2) THEN b2;
END
```

Vidimo, da diagram prehajanja stanj vnašamo v program tako, da specificiramo vsako stanje posebej, po izhodni črki in po vseh prehajanjih stanja. Rezultat, ki ga dobimo v primeru drugega izvornega programa, je enak rezultatu, ki smo ga že spoznali.

10.4.2 Programiranje Mealyjevega avtomata

Kako pridemo do Mealyjevega avtomata na osnovi PPAL-čipov, ne more biti dosti drugače od tega, kar smo videli že pri Mooreovem avtomatu. Vzemimo Mealyjev avtomat na sliki 10.4 (glej razdelek 10.5). Iz začetnega stanja a_0 lahko na osnovi vhodnih dogodkov prispemo v stanje a_4. Iz tega stanja ne moremo z nobenim vhodnim dogodkom več, zato je to stanje

terminalno. Značilnost primera avtomata na sliki 10.4 je, da imamo prehod, na primer, iz stanja a_1 v stanje a_2 pri vseh vhodnih črkah. Zadnje pomeni, da, kakor hitro smo v stanju a_1, gremo tudi v stanje a_2, pri tem pa imamo izhodno črko z_2. Namesto štirih poti med a_1 in a_2 bi lahko narisali samo eno in sicer $1 \,/\, z_2 = (x_1 \vee x_2 \vee x_3 \vee x_4) \,/\, z_2$. Tega pa ne bi mogli napraviti, če bi imeli pri različnih vhodnih črkah različne izhodne črke. Tudi v našem primeru pustimo vse štiri poti izražene (v dani obliki).

Avtomat ima naslednje abecede

$$X = \{x_1, x_2, x_3, x_4\}$$

$$Y = \{a_1, a_2, a_3, a_4, a_5\}$$

$$Z = \{z_1, z_2, z_3, z_4, z_5\}.$$

Za izvorni program potrebujemo tabele kodiranja. Za notranja stanja in izhodne črke imamo

a	q_0	q_1	q_2
a_0	0	0	0
a_1	0	0	1
a_2	0	1	0
a_3	0	1	1
a_4	1	0	0

z	f_0	f_1	f_2
z_0	0	0	0
z_1	0	0	1
z_2	0	1	0
z_3	0	1	1
z_4	1	0	0

.

Za vhodno abecedo nimamo kodirne tabele, zato moramo na vhode dati vsako črko posebej. Lahko pa bi takšno tabelo napravili in s tem prihranili 2 vhoda. Izvorni program za dani Mealyjev avtomat, ob gornjih tabelah, je naslednji:

```
MODULE mealy
 x1 pin 2;
 x2 pin 3;
 x3 pin 4;
 x4 pin 5;
 q2 pin 12 ISTYPE 'reg';
 q1 pin 13 ISTYPE 'reg';
 q0 pin 14 ISTYPE 'reg';
 f2 pin 15 ISTYPE 'reg';
 f1 pin 16 ISTYPE 'reg';
```

```
 f0 pin 17 ISTYPE 'reg';
 sreg = [q2, q1, q0];
 a0 = !q2 * !q1 * !q0;
 a1 = !q2 * !q1 * q0;
 a2 = !q2 * q1 * !q0;
 a3 = !q2 * q1 * q0;
 a4 = q2 * !q1 * !q0;
 na1 = a0 * x1 # a0 * x2 # a0 * x3 # a0 * x4;
 na2 = a1 * x1 # a1 * x2 # a1 * x3 # a1 * x4 # a3 * x4;
 na3 = a2 * x1 # a2 * x2;
 na4 = a3 * x3 # a4 * x1 # a4 * x2 # a4 * x3;
 nz1 = a0 * x1 # a0 * x2 # a0 * x3 # a0 * x4;
 nz2 = a1 * x1 # a1 * x2 # a1 * x3 # a1 * x4;
 nz3 = a2 * x1;
 nz4 = a2 * x2;
 nz5 = a3 * x4;
EQUATIONS
 q2 := na4;
 q1 := na2 # na3;
 q0 := na1 # na3;
 f2 := nz5;
 f1 := nz3 # nz5;
 f0 := nz2 # nz4;
END
```

Z `nai` označujemo naslednje stanje in z `nzi` naslednjo izhodno črko. Avtomat potrebuje šest celic z registrskim načinom delovanja.

Izvorni program Mealyjevega avtomata, ki upošteva vhod v obliki diagrama prehajanja stanj, je

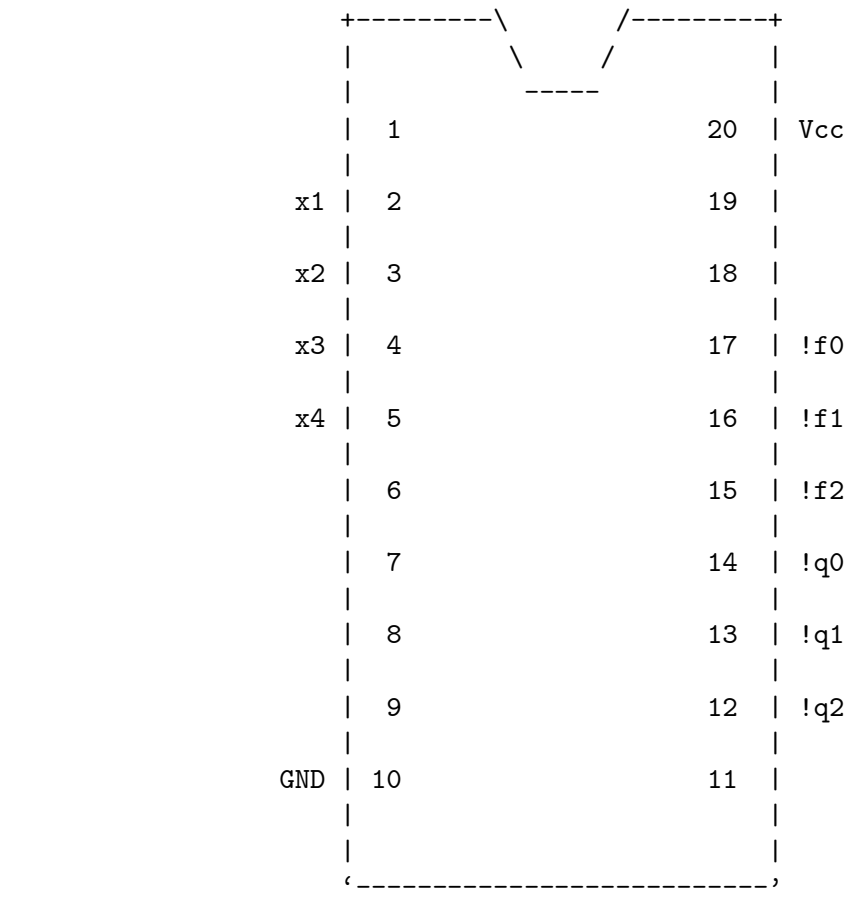
```
MODULE mealy2
 x1 pin 2;
 x2 pin 3;
 x3 pin 4;
 x4 pin 5;
 q2 pin 12 ISTYPE 'reg';
 q1 pin 13 ISTYPE 'reg';
 q0 pin 14 ISTYPE 'reg';
 f2 pin 15 ISTYPE 'reg';
 f1 pin 16 ISTYPE 'reg';
```

```
 f0 pin 17 ISTYPE 'reg';
 sreg = [q2, q1, q0];
 out = [f2, f1, f0];
 a0 = 0; a1 = 1; a2 = 2; a3 = 3; a4 = 4;
 z1 = 0; z2 = 1; z3 = 2; z4 = 3; z5 = 4;
state_diagram sreg;
  State a0:
     IF (x1) THEN a1 WITH out := z1;
     ELSE IF (x2) THEN a1 WITH out := z1;
     ELSE IF (x3) THEN a1 WITH out := z1;
     ELSE IF (x4) THEN a1 WITH out := z1;
  State a1:
     IF (x1) THEN a2 WITH out := z2;
     ELSE IF (x2) THEN a2 WITH out := z2;
     ELSE IF (x3) THEN a2 WITH out := z2;
     ELSE IF (x4) THEN a2 WITH out := z2;
  State a2:
     IF (x1) THEN a3 WITH out := z3;
     ELSE IF (x2) THEN a3 WITH out := z4;
  State a3:
     IF (x3) THEN a4 WITH out := z5;
     ELSE IF (x4) THEN a2;
  State a4:
     IF (x1) THEN a4;
     ELSE IF (x2) THEN a4;
     ELSE IF (x3) THEN a4;
END
```

Za ta primer izvornega programa imamo poročilo `mealy2.rep`

```
                                                          Page 1
ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:14 1998
------------------------------------------------------------
Module                   : 'mealy2'
------------------------------------------------------------
Input files:

   ABEL PLA file         : mealy2.tt3
   Device library        : P16V8R.dev
```

```
Output files:

    Report file          : mealy2.rep
    Programmer load file : mealy2.jed

------------------------------------------------------------
                                                      Page 2
ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:14 1998
P16V8R Programmed Logic:
------------------------------------------------------------
f2.D = !(  !q2.FB & q1.FB & q0.FB & x3 ); '' ISTYPE 'INVERT'
f1.D = !(  !q2.FB & q1.FB & !q0.FB & x1
        #  !q2.FB & q1.FB & !q0.FB & x2 ); '' ISTYPE 'INVERT'
f0.D = !(  !q2.FB & !q1.FB & q0.FB & x1
        #  !q2.FB & !q1.FB & q0.FB & x2
        #  !q2.FB & q1.FB & !q0.FB & !x1 & x2
        #  !q2.FB & !q1.FB & q0.FB & x3
        #  !q2.FB & !q1.FB & q0.FB & x4 ); '' ISTYPE 'INVERT'
q0.D   = ( q2.FB
        #  q0.FB
        #  q1.FB & !x1 & !x2
        #  !x1 & !x2 & !x3 & !x4 ); '' ISTYPE 'INVERT'
q1.D   = ( q2.FB
        #  !q1.FB & !q0.FB
        #  !q0.FB & !x1 & !x2
        #  q1.FB & q0.FB & x3
        #  q1.FB & q0.FB & !x4
        #  !x1 & !x2 & !x3 & !x4 ); '' ISTYPE 'INVERT'
q2.D   = !( q2.FB & !q1.FB & !q0.FB & x1
        #  q2.FB & !q1.FB & !q0.FB & x2
        #  !q2.FB & q1.FB & q0.FB & x3
        #  q2.FB & !q1.FB & !q0.FB & x3 ); '' ISTYPE 'INVERT'
                                                      Page 3
ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:15 1998
P16V8R Chip Diagram:
------------------------------------------------------------
                                    P16V8R
```



SIGNATURE: N/A

ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:15 1998

P16V8R Resource Allocations:

Device Resources	Resource Available	Design Requirement	Unused
Input Pins:			
Input:	10	4	6 (60 %)
Output Pins:			

```
                    |           |             |
        In/Out:     |      8    |      6      |    2 ( 25 %)
        Output:     |      -    |      -      |    -
                    |           |             |
Buried Nodes:       |           |             |
                    |           |             |
        Input Reg:  |      -    |      -      |    -
        Pin Reg:    |      8    |      6      |    2 ( 25 %)
        Buried Reg: |      -    |      -      |    -
                                                          Page 5
ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:15 1998
P16V8R Product Terms Distribution:
-----------------------------------------------------------
            Signal       |    Pin    | Terms | Terms | Terms
             Name        | Assigned  | Used  |  Max  | Unused
=========================|===========|=======|=======|=======
f2.REG                   |   15      |   1   |   8   |   7
f1.REG                   |   16      |   2   |   8   |   6
f0.REG                   |   17      |   5   |   8   |   3
q0.REG                   |   14      |   4   |   8   |   4
q1.REG                   |   13      |   6   |   8   |   2
q2.REG                   |   12      |   4   |   8   |   4

     ==== List of Inputs/Feedbacks ====
Signal Name                          | Pin       | Pin Type
=====================================|===========|=========
x1                                   |    2      | INPUT
x2                                   |    3      | INPUT
x3                                   |    4      | INPUT
x4                                   |    5      | INPUT
                                                          Page 6
ABEL 6.10 - Device Utilization Chart Thu Apr 9 10:50:15 1998
P16V8R Unused Resources:
-----------------------------------------------------------
 Pin   |  Pin   |   Product   | Flip-flop
Number |  Type  |   Terms     |   Type
=======|========|=============|==========
    6  |  INPUT |      -      |    -
    7  |  INPUT |      -      |    -
```

```
 8  |  INPUT |      -      |    -
 9  |  INPUT |      -      |    -
18  |  BIDIR | NORMAL  7   |    D
19  |  BIDIR | NORMAL  7   |    D  . \QTR{tt}{\ }
```

Pri tem poročilu imamo JEDEC-program za programator čipa P16LV8R

```
ABEL 6.10 Data I/O Corp.
JEDEC file for:  P16V8R V9.0
Created on:  Thu Apr 9 10:50:15 1998
*
QP20* QF2194* QV0* F0*
 X0*
NOTE Table of pin names and numbers*
NOTE PINS x1:2 x2:3 x3:4 x4:5 q2:12 q1:13 q0:14 f2:15 f1:16
f0:17*
L0512 01111111111111111111110111101110*
L0544 11110111111111111111110111101110*
L0576 10110111111111111111111011011110*
L0608 11111111011111111111110111101110*
L0640 11111111111101111111110111101110*
L0768 01111111111111111111111011011110*
L0800 11110111111111111111111011011110*
L1024 11111111011111111111110111011110*
L1280 11111111111111111111111111111101*
L1312 11111111111111111111110111111111*
L1344 10111011111111111111111111011111*
L1376 10111011101110111111111111111111*
L1536 11111111111111111111111111111101*
L1568 11111111111111111111111011101111*
L1600 10111011111111111111111011111111*
L1632 11111111011111111111110111011111*
L1664 11111111111101111111110111011111*
L1696 10111011101110111111111111111111*
L1792 01111111111111111111111011101101*
L1824 11110111111111111111111011101101*
L1856 11111111011111111111110111011110*
L1888 11111111011111111111111011101101*
L2048 00111001*
```

```
L2128 11111111111111111111111111111111
      11111111111111111111111111111111*
L2193 1*
C5298*
 F51F
```

10.5 APARATURNI PRISTOP K DIAGRAMU POTEKA

10.5.1 Uvod

Prva faza postavljanja računalniškega postopka je običajno diagram poteka. Iz preteklosti smo vajeni, da podani diagram poteka programiramo v smislu ustreznega izvajalnega programa. V tem primeru govorimo o programski postavitvi računalniškega postopka. Glede na nadaljnji razvoj in današnje stanje računalništva pa prihaja v poštev tudi aparaturno postavljanje postopka. Takšno postavljanje postopka je zlasti pomembno, ko gre za veliko amortizacijo (frekvenco) procesiranja postopka. Če se program izvede samo enkrat ali nekajkrat na dan, je vedno boljše postopek realizirati na programski način. Pri tisoč ali večkratnem procesiranju na dan pa postane programsko postavljena realizacija vprašljiva. Namreč, aparaturna realizacija postopka zahteva pri procesiranju kratek procesorski čas, lahko tudi 10-krat ali 100-krat krajšega od programske realizacije. Pri veliki amortizaciji izvajanja postopka lahko tako prihranimo veliko procesorskega časa, ki je navsezadnje tržno naravnana postavka. Po drugi strani imamo tudi računalniške sisteme, ki delujejo v realnem času. Pri teh je vedno borba s časom, saj mora biti procesiranje končano v naprej predpisanem času. Torej vzrokov za aparaturno postavljanje postopkov je dovolj. Ni rečeno, da pri aparaturni realizaciji ne bomo tudi programirali. Programiranje pa je seveda drugačno: če uporabimo programirne gradnike, programiramo v okviru orodij ABEL, CUPL, A+PLUS, PALASM itd. in ne pa morda v nekem znanem visoko nivojskem jeziku, po katerem praviloma posegajo programerji (izjema je jezik C).

Diagram poteka je pravzaprav univerzalni jezik z razmeroma majhnim številom različnih osnovnih gradnikov. Takšni gradniki so:

- izvajalni operator A (data operation element)
- operator klicanja podrutine K (subroutine call element)
- logični operator p (logic element)

- ▶ terminalni element T
- ▶ začetni element Z
- ▶ element končanja S.

Terminalni element T je samo formalno element in ga potrebujemo le pri podajanju (risanju) diagrama poteka. Z njim gremo lahko iz ene strani na drugo stran zapisa - pri procesiranju pa tega elementa ne potrebujemo. Med naštetimi elementi diagrama poteka ležijo usmerjene poti tako, da diagrame poteka lahko podajamo tudi v obliki matematičnega grafa [19].

Diagram poteka lahko uskladimo z diagrami prehajanja stanj ustreznih avtomatov. Ker se to da napraviti, je diagram poteka lahko predstavljen z množico avtomatov. Potrebni sta dve vrsti avtomatov:

- ▶ izvajalni avtomati,
  ti opravljajo dela, ki so podana z operatorji $A_i, i = 1, 2, ..., q$.
- ▶ nadzorni avtomat,
  ta nadzoruje, da se vse odvija po scenariju diagrama poteka.

V prvi vrsti nas v tem razdelku zanima le nadzorni avtomat.

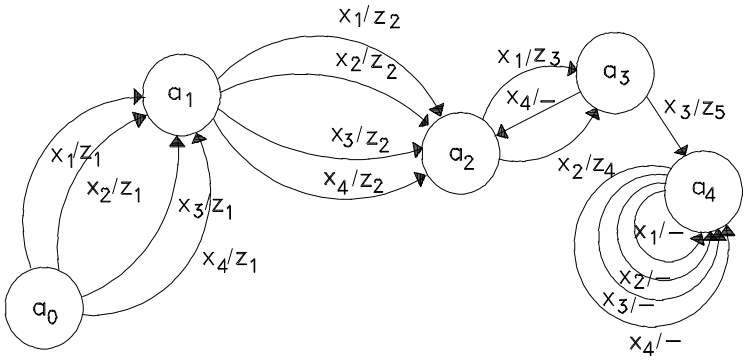
Za razlago prehoda iz diagrama poteka na diagram prehajanja stanj avtomata si poglejmo sliko 10.5. Izvajalni operator A_i nadomestimo s stanjem avtomata a_i. Logični operator p_i ne zahteva stanja, se pa veže na vhodne črke avtomata, ki zahtevajo prehajanja med stanji.

Poznamo veliko iztočnic, kako priti do avtomata, ki nadzoruje potek operacij (procesiranje) po diagramu poteka. Zanimive so predvsem naslednje tri:

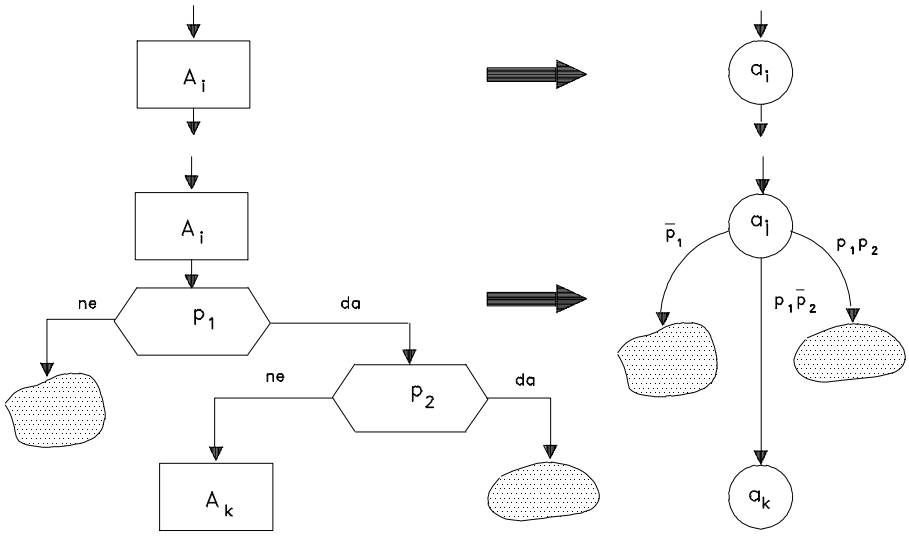
- ∘ Končni avtomat (Mooreov ali Mealyjev avtomat)
- ∘ Wilkesov mikroprogramirni avtomat
- ∘ Avtomat neposredno postavljen v pomnilnik.

Bločno shemo Wilkesovega avtomata vidimo na sliki 10.6. Osnovna zanka je povsem podobna osnovni zanki modela na sliki 10.2. V obeh primerih imamo register R, dekodirnik D in kodirnik K. Razlika pa nastopi pri vhodnih in izhodnih specifikacijah. Izhodna črka z_A je namenjena izvajalnemu operatorju A. Ko je $z_A = 1$ ("enable" signal), lahko omenjeni operator prične s svojim delom. Naslednji operator A' ne more pričeti s svojim delom, če ni $z_{A'} = 1$. Da se to zgodi, se mora v nadzornem avtomatu obstoječe stanje zamenjati z naslednjim stanjem. Vhodni signali x prihajajo iz aktivnega operatorja A in povedo, kakšno je stanje tega operatorja.

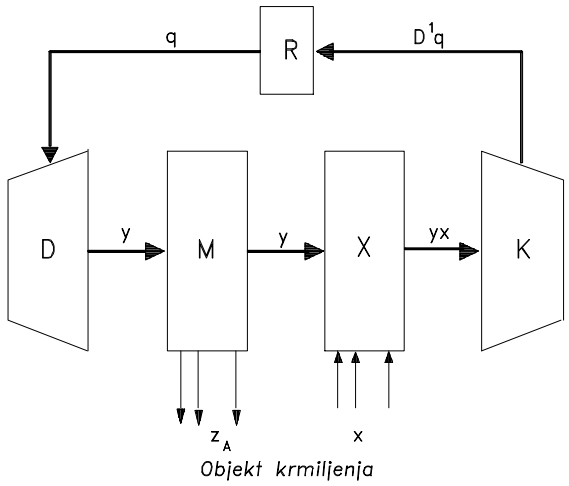
Na sliki 10.7 vidimo avtomat, ki je neposredno vstavljen v pomnilnik POM (v pomnilniku je tabela prehajanja stanj in tabela izhodnih črk) [20]. Na osnovi vhodnega vektorja $\mathbf{x}$ in v pomnilniku shranjenega obstoječega



Slika 10.4: Primer Mealyjevega avtomata.



Slika 10.5: Prehod iz diagrama poteka na diagram prehajanja stanj avtomata.



Slika 10.6: Bločna shema Wilkesovega avtomata.

stanja y izračunamo z odločitvenim preklopnim vezjem OPVA naslov, na katerem leži naslednje stanje in ustrezna izhodna črka. Črke z_A vstopajo v opazovani izvajalni operator A, iz katerega tudi prihaja vektor $\mathbf{x}$.

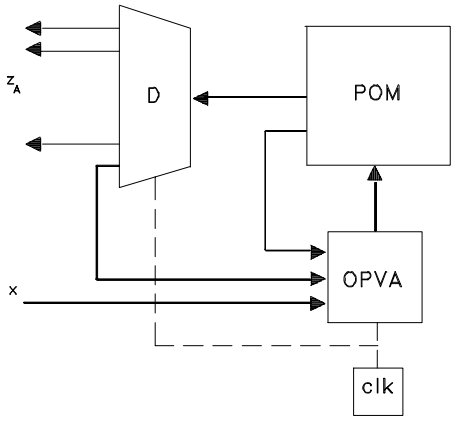
Vse tri navedene možnosti pristopa k diagramu poteka na avtomatni način so prikladne predvsem pri uporabi programirnih gradnikov.

10.5.2 Avtomat za nadzorovanje diagrama poteka

Matrika poteka dogajanj v diagramu poteka ima naslednjo značilno razsežnost

$$M = \begin{array}{c|cccccc} & A_1 & A_2 & . & . & . & A_n \quad S \\ Z & & & & & & \\ A_1 & & & & & & \\ . & & & & & & \\ . & & & f_{ij} & & & \\ . & & & & & & \\ A_n & & & & & & \quad . \end{array} \tag{10.9}$$

V diagramu poteka so v tem primeru izvajalni operatorji $A_i, i = 1, 2, ..., n$.



Slika 10.7: Shollov nadzorni avtomat.

Z je začetno stanje in S končno stanje diagrama poteka. Element f_{ij} matrike M je lahko

1. $f_{ij} = 0$, če ni neposrednega prehoda iz izvajalnega operatorja A_i v operator A_j.
2. $fij = 1$, če za izvajalnim operatorjem A_i sledi operator A_j in
3. $f_{ij} = f(p_1, p_2, ..., p_m)$, če obstaja logični pogoj $f(p_1, p_2, ..., p_m)$, da iz A_i preidemo v izvajalni operator A_j.

V matriki M so izpolnjeni naslednji pogoji:

$f_{ij}\, f_{ik} = 0, \quad j \neq k, \quad$ za vsak par elementov (izven diagonale) v vrstici i

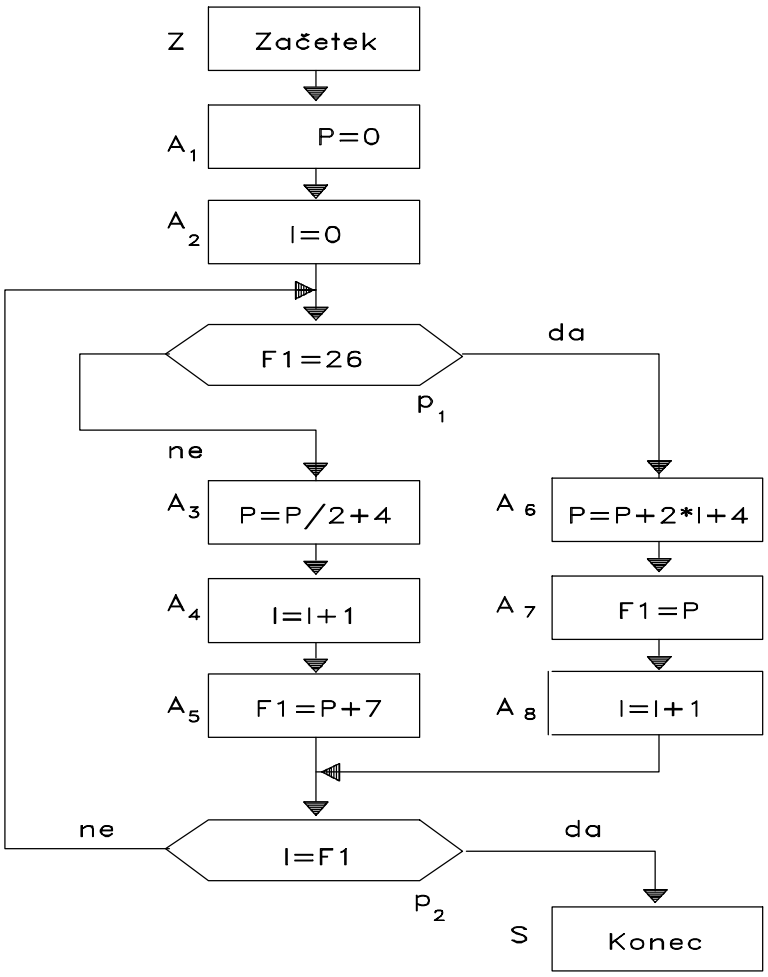
$f_{ii} = e, \quad$ diagonalni elementi

$\bigvee_{1}^{j=n+1} f_{ij} = 1, \quad$ za vsako vrstico i.

Vrednost e pomeni, da operator A ne more prehajati nazaj na lastni vhod brez vmesne kontrole, ki zavisi od pogojev p.

Zgled 12 *:*

Za diagram poteka na sliki 10.8 *napišimo matriko* M.



Slika 10.8: *Diagram poteka označen za prevedbo na diagram prehajanja stanj nadzornega avtomata.*

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e & \bar{p}_1 & 0 & 0 & p_1 & 0 & 0 & 0 \\ 0 & 0 & e & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & e & p_1\bar{p}_2 & 0 & 0 & p_2 \\ 0 & 0 & 0 & 0 & 0 & e & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e & 1 & 0 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & 0 & p_1\bar{p}_2 & 0 & e & p_2 \end{bmatrix} \tag{10.10}$$

(konec zgleda)

Matrika M se nanaša na diagram poteka. S pomočjo preslikav, ki jih vidimo na sliki 10.5, pa pridemo do matrike prehajanja stanj $P(\mathbf{p})$ nadzornega avtomata tako, da izvajalne operatorje A_i zamenjamo s stanji a_i nadzornega avtomata in izpustimo začetni in končni operator Z in S v matriki M. Za matriko (10.10) imamo pri vhodnem vektorju $\mathbf{p} = [p_1, p_2]$

$$P(\mathbf{p}) = \begin{bmatrix} e & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e & \bar{p}_1 & 0 & 0 & p_1 & 0 & 0 \\ 0 & 0 & e & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e & 1 & 0 & 0 & 0 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & e & p_1\bar{p}_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e & 1 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & 0 & p_1\bar{p}_2 & 0 & e \end{bmatrix}. \tag{10.11}$$

Vzemimo stolpčni a_i kot naslednje stanje in vrstični a_j kot sedanje stanje.

Po tej opredelitvi dobimo δ funkcijo

$$\begin{aligned} \mathrm{D}^1 a_2 &= a_1 \\ \mathrm{D}^1 a_3 &= a_2\bar{p}_1 \vee a_5\bar{p}_1\bar{p}_2 \vee a_8\bar{p}_1\bar{p}_2 \\ \mathrm{D}^1 a_4 &= a_3 \\ \mathrm{D}^1 a_5 &= a_4 \\ \mathrm{D}^1 a_6 &= a_2 p_1 \vee a_5 p_1\bar{p}_2 \vee a_8 p_1\bar{p}_2 \\ \mathrm{D}^1 a_7 &= a_6 \\ \mathrm{D}^1 a_8 &= a_7. \end{aligned} \tag{10.12}$$

Stanja $a_i, i = 1, 2, ..., 8$ zakodiramo in s tem preidemo iz osmih stanj na tri pomnilne celice oziroma na vektor $\mathbf{q}$. Tudi vhodne pogoje lahko zakodiramo in s tem preidemo na primer iz pogojev $\mathbf{p}$ na vektor $\mathbf{x}$. Tako imamo

a_i	q_1	q_2	q_3	q_4
a_1	0	0	0	0
a_2	0	0	1	0
a_3	0	1	0	0
a_4	0	1	1	0
a_5	1	0	0	0
a_6	1	0	1	0
a_7	1	1	0	0
a_8	1	1	1	0
z	0	0	0	1
s	0	0	0	1

p_1	p_2	$\mathbf{x}$
0	0	x_1
0	1	x_2
1	0	x_3
1	1	x_4

.

V kodirni tabeli za stanja imamo tudi četrto pomnilno celico z izhodom q_4. Ta celica je tekom delovanja nadzornega avtomata postavljena v stanje 0, na začetku in koncu diagrama poteka pa v stanje 1 (stanje 'ready'). Tako je sistem začetka in konca izvajanja diagrama poteka (programa) logično ločen od notranjosti diagrama poteka. Takšno ločitev smo predvideli že pri prehodu iz matrike diagrama poteka $M(\mathbf{p})$ na matriko prehajanja stanj $P(\mathbf{p})$. Prednost je v tem, da ima matrika $P(\mathbf{p})$ manjši red kot $M(\mathbf{p})$.

Tako smo prišli do vseh postavk, ki so potrebne, da po Iversonovi notaciji izračunamo krmilne funkcije za register R

$$\mathrm{D}^1\mathbf{q} = (\mathbf{q}\& \equiv D') \vee \&P(\mathbf{p}) \vee \&D\,. \tag{10.13}$$

Matriko D dobimo iz gornje tabele kodiranja pri q_1, q_2 in q_3

$$D = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

Izraz (10.13) dobimo na osnovi modela, ki je podan s sistemom (10.8), zato ne potrebuje posebnega komentarja.

Zgled 13 *:*

Za *diagram poteka na sliki* 10.8 *izračunajmo po izrazu* (10.13) *krmilne funkcije pomnilnih celic registra* R.

Najprej si izračunajmo zadnji del izraza (10.13)

$$P \vee \&D \;=\; \begin{bmatrix} e & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & e & \bar{p}_1 & 0 & 0 & p_1 & 0 & 0 \\ 0 & 0 & e & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & e & 1 & 0 & 0 & 0 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & e & p_1\bar{p}_2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & e & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & e & 1 \\ 0 & 0 & \bar{p}_1\bar{p}_2 & 0 & 0 & p_1\bar{p}_2 & 0 & e \end{bmatrix} \vee \& \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 1 \\ p_1 & \bar{p}_1 & p_1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ p_1\bar{p}_2 & \bar{p}_1\bar{p}_2 & p_1\bar{p}_2 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ p_1\bar{p}_2 & \bar{p}_1\bar{p}_2 & p_1\bar{p}_2 \end{bmatrix}.$$

In prvi del izraza (10.13)

$$\mathbf{q}\& \equiv D' = [q_1, q_2, q_3] \vee \& \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} =$$

$$[\bar{q}_1\bar{q}_2\bar{q}_3, \bar{q}_1\bar{q}_2 q_3, \bar{q}_1 q_2\bar{q}_3, \bar{q}_1 q_2 q_3, q_1\bar{q}_2\bar{q}_3, q_1\bar{q}_2 q_3, q_1 q_2\bar{q}_3, q_1 q_2 q_3] =$$

$$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8]\,.$$

Oba dela izraza skupaj nam dasta krmilne funkcije

$$\left[\mathrm{D}^1 q_1, \mathrm{D}^1 q_2, \mathrm{D}^1 q_3\right] =$$

$$[a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8] \vee \&$$

$$\vee\& \begin{bmatrix} 0 & 0 & 1 \\ p_1 & \bar{p}_1 & p_1 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ p_1\bar{p}_2 & \bar{p}_1\bar{p}_2 & p_1\bar{p}_2 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ p_1\bar{p}_2 & \bar{p}_1\bar{p}_2 & p_1\bar{p}_2 \end{bmatrix}$$

$$\begin{aligned} \mathrm{D}^1 q_1 &= a_4 \vee a_6 \vee a_7 \vee a_2 \vee a_2 p_1 \vee a_5 p_1\bar{p}_2 \vee a_8 p_1\bar{p}_2 \\ \mathrm{D}^1 q_2 &= a_3 \vee a_6 \vee a_7 \vee a_2 \vee a_2\bar{p}_1 p_2 \vee a_5\bar{p}_1\bar{p}_2 \vee a_8\bar{p}_1 p_2 \\ \mathrm{D}^1 q_3 &= a_1 \vee a_3 \vee a_7 \vee a_2 \vee a_2 p_1 \vee a_5 p_1\bar{p}_2 \vee a_8 p_1\bar{p}_2. \end{aligned}$$

(konec zgleda)

Manjka nam le še krmilni signal, ki določa čas izvajanja operacij v posameznih izvajalnih operatorjih. V diagram poteka na sliki 10.8 bi morali sedaj vpisati še izhodne črke avtomata z_{A_i}, $i = 1, 2, ..., 8$. Operator A_i lahko izvaja svoje operacije le v primeru, ko je izhodna črka $z_{A_i} = 1$. V primeru ko istočasno ne more biti več kot ena izhodna črka po vrednosti 1, izvaja svoje operacije lahko le en izvajalni operator. Ko je na primer $z_{A_i} = z_{A_j} = 1$, sta lahko aktivna oba operatorja A_i in A_j (paralelnost v izvajanju diagrama poteka).

Da pridobimo na času procesiranja, pregledamo diagram poteka s stališča možnega paralelnega delovanja izvajalnih operatorjev. Paralelno so k delovanju lahko vzbujani samo tisti izvajalni oparatorji, ki so mad seboj neodvisni. Za diagram poteka na sliki 10.8 imamo tako

$$\begin{aligned} z_1 &= z_{A_1} = z_{A_2} \\ z_2 &= z_{A_3} = z_{A_4} \\ z_3 &= z_{A_5} \\ z_4 &= z_{A_6} \\ z_5 &= z_{A_7} = z_{A_8}. \end{aligned} \tag{10.14}$$

ali pa

$$\begin{aligned} z_1 &= z_{A_1} = z_{A_2} \\ z_2 &= z_{A_3} \\ z_3 &= z_{A_5} = z_{A_5} \\ z_4 &= z_{A_6} \\ z_5 &= z_{A_7} = z_{A_8}. \end{aligned} \tag{10.15}$$

Iz sistema izrazov (10.14) dobimo izhodno matriko I. Ta nam da odnos med stanji oziroma izvajalnimi operatorji ter različnimi izhodnimi črkami. Tako imamo matriko reda 8×5

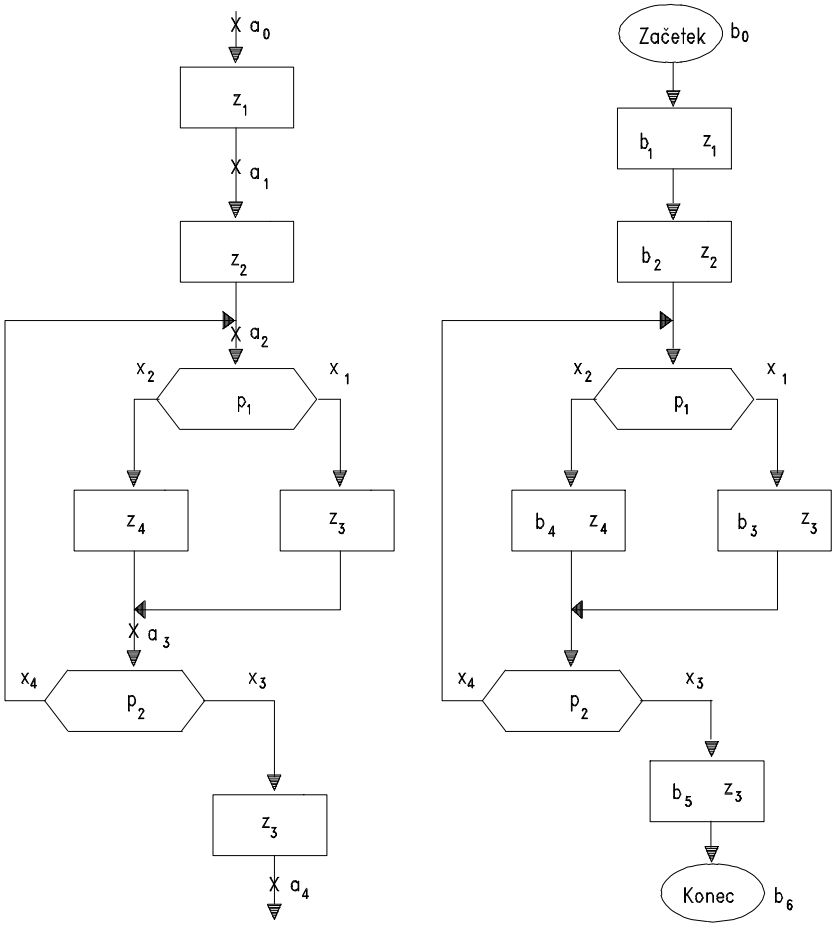
$$I = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \\ A_5 \\ A_6 \\ A_7 \\ A_8 \end{array} \begin{array}{c} \begin{array}{ccccc} z_1 & z_2 & z_3 & z_4 & z_5 \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}. \tag{10.16}$$

Ker izhodne črke z zavisijo samo od stanj a oziroma od izvajalnih operatorjev A, je dobljeni nadzorni avtomat Mooreov avtomat. Kako tak avtomat realiziramo s PPAL-gradniki, smo si že ogledali.

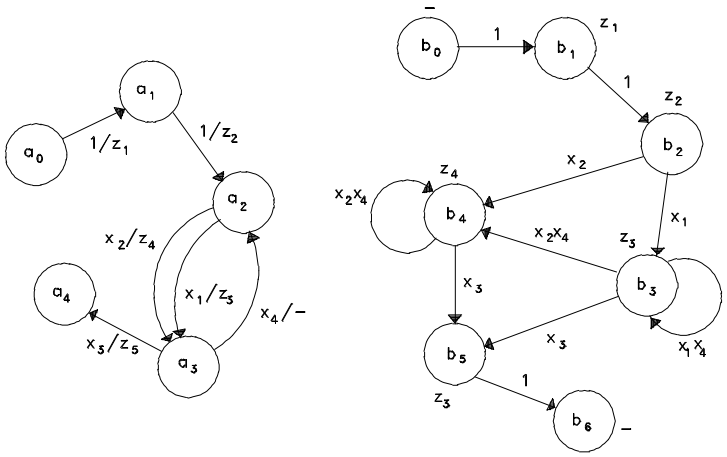
10.5.3 Neposredno Mealyjevo in Mooreovo označevanje diagrama poteka

Diagram poteka lahko označimo z dodatnimi informacijami tako, da je možen takojšen prehod iz diagrama poteka na diagram ali tabelo prehajanja stanj nadzornega avtomata. Na sliki 10.9 vidimo diagram poteka, ki je Mealyjevsko in Mooreovsko označen. Prvi primer označitve je na levi in drugi na desni strani slike. Aplikacijske označitve izvajalnih in logičnih operatorjev na sliki niso podane.

Ker je značilnost Mealyjevega avtomata v tem, da je izhodna črka na prehodu med dvema stanjema, damo stanje na povezavi pred in po izvajalnem operatorju, izhodno črko pa dodelimo operatorju, ki leži med povezavama. Vhodne kanale p lahko opazujemo v smislu vhodnih črk x na



Slika 10.9: Mealyjevo in Mooreovo označevanje diagrama poteka.



Slika 10.10: Mealyjeva (levo) in Mooreova označitev (desno) diagrama poteka.

enak način tako pri Mealyjevem kot pri Mooreovem avtomatu. Če je $p = 0$ imamo x_1, in če je $p = 1$, imamo x_2. Pri Mooreovem avtomatu imamo (stacionarno) izhodno črko pri stanju, v katerem se nahaja. Zato dodelimo tako stanje kot tudi ustrezno izhodno črko izvajalnemu operatorju. Ker je prva izhodna črka pri Mooreovem avtomatu (glede na ekvivalenco z Mealyjevim avtomatom) neuporabna pri vhodnih dogodkih, vstavimo začetno in končno stanje b_0 in b_6.

Na sliki 10.10 vidimo diagrama prehajanja stanj za ustrezni Mealyjev avtomat (levo) in Mooreov avtomat (desno). Vhodna črka 1 pomeni, da se prehod iz stanja v stanje dogodi pri vseh vhodnih črkah oziroma ne glede na vhodno črko. Stanje b_1 gre brezpogojno v stanje b_2, če le izvajalni operator, na katerega je vezano stanje b_1, konča s svojim delom. Izhodna črka $z = -$ pomeni, da nimamo v opazovanem primeru nobenega izvajanja izvajalnih operatorjev v diagramu poteka.

Če se v diagramu poteka nahaja več enakih izvajalnih operatorjev, lahko pri Mooreovi označitvi uporabimo za vse enako izhodno črko. Ne smemo pa uporabiti enako stanje avtomata, ker se, čeprav gre za enako izvajanje, dogodki odvijajo v različnih časih. Tak primer vidimo na sliki 10.10 v primeru parov (b_3, z_3) in (b_5, z_3). Torej imamo v obeh primerih isti izvajalni operator. Tudi logični pogoji so v istem diagramu poteka enaki. Vse enake

logične pogoje navežemo na isto oziroma iste vhodne črke.

Mooreov avtomat ima več ali kvečjemu enako število stanj kot ekvivalentni Mealyjev avtomat. Dva avtomata pa sta ekvivalentna, če imata pri istih vhodnih dogodkih enake izhodne dogodke.

Ko pridemo do diagramov prehajanja stanj za Mealyjev in Mooreov avtomat v smislu slike 10.10, delo še ni končano. Možno je namreč izpeljati minimizacijo avtomatov, kar nam da najmanjše število pomnilnih celic in najenostavnejše krmiljenje le-teh. Iz razpoložljive vsebine PPAL-gradnikov pa izhaja, do kakšne mere je minimizacija sploh potrebna.

10.5.4 Postavitev avtomata v pomnilnik

V okviru slike 10.7 smo omenili, da lahko avtomat postavimo neposredno v delovni pomnilnik. Pri takšnem načrtovanju avtomata nam je potrebno tabelo prehajanja stanj $P(\mathbf{x})$ in izhodno tabelo $I(\mathbf{x})$ vstaviti v delovni pomnilnik in zagotoviti nemoten takojšen dostop do teh tabel (na osnovi vhodnih in izhodnih instrukcij). Pri tem sta pomembni predvsem dve postavki: velikost potrebnega pomnilnika in velikost odločitvenega vezja za izračun naslova naslednjega stanja, OPVA. Večkrat takšne avtomate vstavljamo v majhne mikroračunalniške sisteme, pri katerih ni veliko možnosti niti za prvo in niti za drugo omenjeno postavko.

Izkaže se, da če imamo bogate možnosti v delovnem pomnilniku, imamo lahko enostavne enote OPVA - seveda na račun redundance v pomnilniku. Če pa imamo malo prostora v pomnilniku, je postavitev še vedno možna - vendar na račun kompleksnosti OPVA. Postavitev avtomata v pomnilnik je vedno kompromis med redundanco v pomnilniku in kompleksnostjo OPVA. Kaj pomeni prvo in kaj drugo? Odgovor na to je poskušal dati Sholl [20], po katerem povzemamo naslednje.

Redundanca pomnilnika (tabele v pomnilniku) je

$$R = \frac{\text{neuporabljena kapaciteta pomnilnika}}{\text{celotna kapaciteta pomnilnika}}\,. \tag{10.17}$$

Decizijsko vezje (v splošnem in ne samo v primeru OPVA) ima kompleksnost

$$C = \sum_{i=1}^{I} \frac{s(f_i)}{I\,V}\,. \tag{10.18}$$

Pri tem je s število neodvisnih spremenljivk, ki nastopijo v logični funkciji f_i. I je število vseh funkcij vezja in V število vseh vhodnih spremenljivk vezja.

Zgled 14 *:*

Izračunajmo kompleksnost sistema logičnih funkcij

$$\begin{aligned} f_1(x_1, x_2, x_3) &= x_2 x_3 \\ f_1(x_1, x_2, x_3) &= x_1 \vee x_2 \vee x_3 \\ f_1(x_1, x_2, x_3) &= x_1 \bar{x}_2 x_3 \,. \end{aligned} \tag{10.19}$$

Na osnovi izraza (10.18) *dobimo*

$$C = \frac{2+3+3}{3 \cdot 3} = \frac{8}{9}\,.$$

Maximalna kapaciteta vezja je

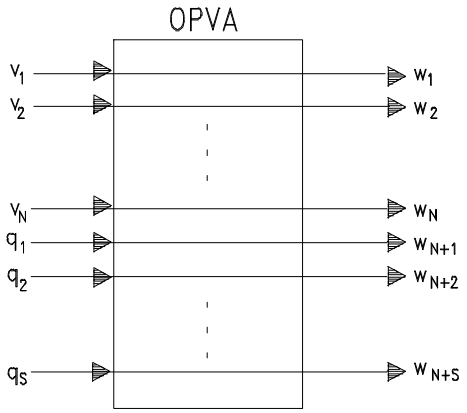
$$C_{\max} = \frac{V + V + ... + V \ (I\text{-krat})}{I\,V} = 1 \tag{10.20}$$

In minimalna

$$C_{\min} = \frac{1 + 1 + ... + 1 \ (I\text{-krat})}{I\,V} = \frac{I}{I\,V} = \frac{1}{V} \tag{10.21}$$

Kompleksnost ima tako vrednost iz področja $1/V \leq C \leq 1$, medtem ko imamo pri redundanci v pomnilniku $0 \leq R \leq 1$.

Vzemimo, da ima matrika prehajanja stanj dimenzijo $n \times r$. Pri tem je n število vhodnih spremenljivk in r število stanj. Število ustreznih dvojiških kanalov je $N = [log_2(n)]$ in $S = [log_2(p)]$ (oglati oklepaj pomeni prvo višje ali enako celo število). OPVA ima tako število vhodov $V = S + N$. Pri najenostavnejši logiki OPVA, to vidmo na sliki 10.11, imamo tudi število izhodov $I = S + N$. V tem primeru imamo kompleksnost vezja in pomnilno redundanco



Slika 10.11: Najenostavnejše vezje OPVA.

$$C = \frac{N+S}{(N+S)(N+S)} = \frac{1}{N+S} = \frac{1}{V}$$
$$R = \frac{2^{N+1}-c}{2^{N+1}} = 1 - \frac{c}{2^V}\,. \tag{10.22}$$

Pri tem je c je število zasedenih mest v tabeli prehajanja stanj avtomata. Če avtomat minimiziramo, dobimo manjšo tabelo prehajanja stanj, kar nam da manjši C in R.

Na sliki 10.11 imamo število izhodov $I = N + S$ označene z w. V kolikor v OPVA uvajamo logične operatorje, se nam število izhodov w_i zmanjša glede na število vhodov $V = N + S$. V tem primeru se nam ustrezno poveča kompleksnost C in zmanjša redundanca R.

Zgled 15 *:*

Dan je Mooreov avtomat s tabelo prehajanja stanj

vhod / stanje		$\bar{v}_1\bar{v}_2\bar{v}_3$	$\bar{v}_1\bar{v}_2v_3$	$\bar{v}_1v_2\bar{v}_3$	$\bar{v}_1v_2v_3$	$v_1\bar{v}_2\bar{v}_3$
		x_1	x_2	x_3	x_4	x_5
$\bar{q}_1\bar{q}_2$	b_1	-	-	-	-	6
$\bar{q}_1q_2$	b_2	1	-	3	-	-
$q_1\bar{q}_2$	b_3	-	2	-	-	-
q_1q_2	b_4	-	-	4	5	-

Izračunajmo C in R *pri dani razsežnosti matrike prehajanja stanj. Enako napravimo, če matriko zmanjšamo tako, da je redundanca* $R = 0$. *V tabelo je vpisana adresa celice, v kateri se nahaja naslednje stanje.*

Iz tabele izhajajo podatki oziroma izračuni le-teh

$n = 5$

$N = [log_2 5] = 3$

$s = 4$

$S = [log_2 4] = 2$

$V = N + S = 3 + 2 = 5$

$c = 6.$

Za primer slike 10.11 *imamo*

$$C = \frac{1}{5}$$

$$R = 1 - \frac{6}{25} = \frac{13}{16}\,.$$

Pri redundanci $R = 0$ *imamo manjšo tabelo prehajanja stanj, ki jo lahko konfiguriramo z izhodi* w, *kot na primer*

izhodi	w_1	$\bar{w}_1$
$\bar{w}_2\bar{w}_3$	1	4
$\bar{w}_2w_3$	2	5
$w_2\bar{w}_3$	3	6

Iz Veitchevih diagramov za

$$w_1(v_1, v_2, v_3, q_1, q_2)$$
$$w_2(v_1, v_2, v_3, q_1, q_2)$$
$$w_3(v_1, v_2, v_3, q_1, q_2)$$

dobimo

$$w_1 = \bar{q}_1 q_2 \vee q_1 \bar{q}_2$$

$$w_2 = v_2 \bar{q}_1 \vee v_1$$

$$w_3 = v_2 \,.$$

Vidimo, da imamo $s(w_1) = 2$, $s(w_2) = 3$, $s(w_3) = 1$. *Zadnje nam da pri* $R = 0$ *kompleksnost za* OPVA

$$C = \frac{2}{3 \cdot 5} + \frac{3}{3 \cdot 5} + \frac{1}{3 \cdot 5} = \frac{2}{5} \,.$$

Poglavje 11

SPECIFIKACIJA IN OPISOVANJE RSS

11.1 UVOD

Zakaj želimo načrtovanje RSS dopolniti tudi s specifikacijo in opisovanjem le-teh? Naš osnovni namen je predstaviti znanja za načrtovanje RSS, vendar načrtovanja ne moremo napraviti, če predhodno ne postavimo dovolj strogo zahteve, katerim mora zadoščasti načrtovani primerek RSS. V ciklu postavljanja RSS imamo naslednje zaokrožene tehnike:

- *tehnika postavljanja zahtev* (requirement engineering), zaključek te tehnike nam da konceptualni model (conceptual model) RSS,
- *tehnika neposrednega načrtovanja* (design engineering), zaključek te tehnike je model načrtovanja (snovanja) (design model) RSS,
- *tehnika postavitve* (realization engineering), zaključek te tehnike je realizicijski model (build model) RSS,
- *tehnika sistemske integracije* (system integration), zaključek te tehnike je proizvodni model sistema (production model) in
- *tehnika neposredne proizvodnje* (production), zaključek te tehnike je postavitev in izvajanje proizvodnje.

Podani spisek tehnik nam nazorno prikazuje, kako je neposredno načrtovanje RSS vpeto v cikel nastajanja RSS. Vidimo torej, da načrtovanje lahko

le sledi določenemu konceptualnemu modelu RSS. Ta model pa zahteva dokaj striktno opisovanje in specifikacijo delovanja načrtovanega sistema. Da bi dosegli dovoljšnjo, takorekoč matematično striktnost, so postavili visoko nivojske programske jezike (vemo, da visoko nivojski pristop zahteva prevajanje), katerih osnovne informacije želimo opredeliti tudi v tem poglavju.

Glede na povedano, v računalništvu poznamo, že kar vrsto let, specifikacijske in opisne jezike. Postavili so osnovno teorijo le-teh, vendar so se v primerjavi z običajnimi programskimi visoko nivojskimi izvajalnimi jeziki, relativno malo uporabljali. Večjo implementacijo so omenjeni jeziki v preteklosti doživeli na področju telekomunikacij in to takoj, ko se je na tem področju začela sistemsko uveljavljati tudi računalniška tehnologija. Tako se je že v letu 1972 pričelo prizadevanje za standardno specifikacijo in opisovanje kompleksnih komunikacijskih sistemov, še zlasti zato, ker so ti praviloma pogojeni z delovanjem v realnem času. V omenjenih letih se je uveljavljal jezik SDL (Specification and Description Language), ki so ga pri CCITT privzeli za svoj standard, akceptiral pa ga je tudi ISO. V letih 1976, 1980, 1984, 1988 so se zvrstile izboljšave (nove verzije) jezika [21], ki so v veliki meri poskrbele za prodor računalništva na področje komunikacij ali, če hočemo, prodor komunikacij na področje računalništva. Danes imamo opraviti že z integriranimi sistemi, pri katerih računalništva in komunikacij niti ne moremo več ločevati. Če ne bi bilo dovoljšnje sinergije med komunikacijami in računalništvom, danes ne bi bili priča Internetu, infocestam, sistemu ISDN, računalniškim mrežam itd.

SDL je, kot standard in kot nestandard, uporaben zlasti v sistemih, ki delajo v realnem času, interaktivnih sistemih (delo s protokoli) in porazdeljenih sistemih. Ključne besede jezika pri načrtovanju sistemov so: nedvomnost, jasnost, preciznost, konciznost, korektnost in konsistentnost, čeprav je sistem, na katerem delamo, velik. Pri podajanju sistemov omenjenih značilnosti naravni jezik nima in je zato neprimerljiv z SDL, ki sodi v okvir formalnih jezikov in deluje natančno kot, recimo, matematični model. Zaradi teh značilnosti si moramo ogledati vsaj fragmente osnovnega dela SDL.

SDL pa ni edino vidno okolje za specifikacijo in načrtovanje sodobnih računalniško-kominikacijskih sistemov. Tako imamo še VHDL, LOTOS, HOOD, AKL in druge. Ker postajajo računalniški sistemi izredno veliki (računalniške mreže, informacijske glavne ceste itd.), postaja vedno bolj pomembno prehajanje iz enega okolja na drugega oziroma integracija okolij. V tem poglavju si oglejmo nekaj značilnosti tudi glede tega problema.

11.2 SDL

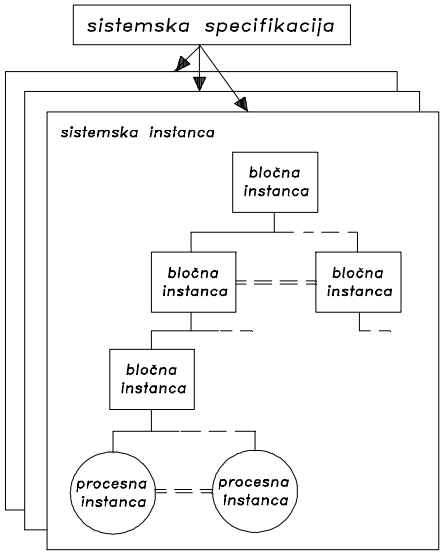
11.2.1 Osnovne sistemske postavke

SDL je delovno okolje, ki je privzelo metode in postopke FSM (Finite State Machine). V SDL sta se zlasti dodatno izpilila pojma odločitev in opravilo (decision, task) tako, da današnji SDL dokaj razširja FSM (EFSM, Extended-State Machine). Enostavno povedano, SDL je okolje, ki se naslanja na osnovno teorijo avtomatov, vendar tako, da so v njem dodatno izdelali vso tehniko, ki je potrebna od abstraktnosti do detajlno tehnično izdelanih procesov delovanja sistema. SDL lahko sistem obdela na nivoju sistema, nivoju blokov in na nivoju procesa sistema. Zlasti je pomembna SDL-tehnika za izmenjavo informacij med procesi v sistemu (koncept protokolov).

Osnovni trije pojmi, ki so temeljno razdelani v SDL, so specifikacija, tip in instanca. Specifikacija je postavka, ki definira tip. Tip je navadno množica objektov, ki imajo enako značilnost ali značilnosti v sistemu in zato sodijo v okvir ene specifikacije. Če vzamemo na primer hiše kot opazovane objekte in hišne številke na njih, je hiša tip, instanca pa na primer hiša s hišno št. 10. V naravnem jeziku velikokrat zamenjujemo naslov in objekt pod tem naslovom, kar na nek način onesposablja naravni jezik, v domeni jasnosti in konciznosti izražanja. Tako, na primer, noben ne pomisli, da je Trg revolucije 1 samo naslov ali številka, ker večina ljudi takoj pomisli na stolpnico LB v Ljubljani ali celo na žiro račun, če ga ima tam v banki.

Pri opisovanju sistemov imamo sistemsko specifikacijo, sistemsko instanco in bločno instanco. Pri tem je instanca nekaj, kar ima odnos navzgor in navzdol. Sistemska instanca je predstavljena z diagramom prehajanja stanj, če so za stanja v tem diagramu privzete različne (lahko tudi enake) procesne instance. V omenjenem diagramu prehajanja stanj je za spremenljivko prehajanja (v avtomatu vhodna črka) privzet signal (signal instance). Iz tega sledi, da ima sistemska instanca svoj potek po času, čas pa upoštevamo asinhronsko. Zadnje pomeni, da pridemo do prehoda iz ene procesne instance v drugo takoj, ko se pojavi signal. Procesni instanci pripada FIFO-vrsta. Ta omogoča, da je procesna instanca v stanju čakanja ali pa v izvajanju prehajanja med dvemi stanji. Če v prispodobi rečemo, da je sistemska specifikacija izvorni program, je sistemska instanca izvajalni program.

Procesna instanca je torej stanje v diagramu prehajanja stanj, ki ga imenujemo sistemska instanca. Procesna instanca pa se zaradi struktrnosti



Slika 11.1: Odnosi med sistemsko specifikacijo in sistemsko instanco.

opazovanega sistema vedno nahaja v neki v bločni instanci. V bločni instanci imamo množico procesnih instanc istega tipa. Dve procesni instanci v bločni instanci sta v medsebojni relaciji, ki jo omogoča pot signala med obema instancama (signal route instance). V sistemski instanci je navadno več bločnih instanc. Te so v medsebojnem odnosu, ki ga omogoča kanalna instanca.

Na sliki 11.1 vidimo relacijo med sistemsko specifikacijo in sistemsko instanco, viden pa je tudi odnos sistemske instance do bločne ter procesne instance. Kot vidimo, je sistemska specifikacija množica sistemskih instanc. Sistemska instanca je množica tistih blokov, ki so v medsebojnem odnosu, ko gre za procesne instance določenega tipa. Namen sistemske specifikacije je, da na osnovi nje opazujemo vedenje sistema, ki ima svojo strukturo, kreirano po blokih.

Glede na sliko 11.1 lahko rečemo, da je vsak sistem razdeljen na tri nivoje: sistemski nivo, bločni nivo in procesni nivo. Med temi nivoji je vzpostavljena povezava s pretakanjem signalov po kanalih. Signal, ki je lahko poslan ali sprejet, lahko poseduje množico parametrov (vrednosti).

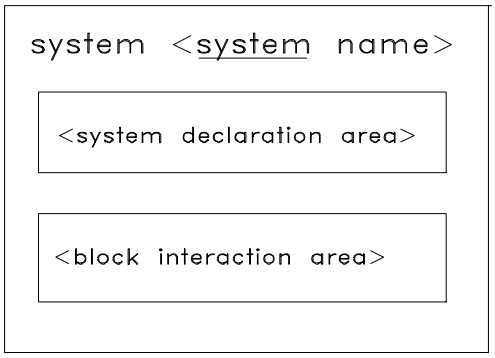
11.2.2 Osnovna sintaksa

SDL je jezik, ki ima, kot vsak drug jezik, svojo sintakso in semantiko. Sintaksa jezika je grafična ali besedna. Besedna sintaksa se nanaša na BNF-koncept (Backus-Naur Form), grafična sintaksa pa je opisana z modificiranim (razširjenim) BNF. Če SDL opazujemo z vidika programskega jezika, je potrebno reči, da je besedilna sintaksa SDL v strogi relaciji s programskim jezikom pascal.

SDL specifikacija je hierarhično postavljena tako, da so komponente specifikacije postavljene drevesno. Komponente so opisane direktno v nivoju ali pa je njihov opis podan od zunaj, izven nivoja. Sistemska specifikacija je lokalna ali referenčna, kjer referenca prihaja od zunaj (remote). Gornji nivo sistemske specifikacije opisuje množico blokov, ki delujejo med seboj in z okoljem. Medsebojno sporazumevanje poteka preko kanalov (tako med bloki kot bloki in okoljem).

Vsaka sistemska specifikacija ima dva ločena dela. Prvi se nanaša na deklaracije tipov podatkov, signalne liste, signale itd. in drugi na interakcije med bloki. Interakcije med bloki definirajo arhitekturo celotne sistemske specifikacije. Grafično in besedno predstavitev omenjenih delov specifikacije vidimo na sliki 11.2 in v besednem zapisu

```
system <system name>;
    <system declaration>
    <block interaction>
endsystem [<system name>];
```



Slika 11.2: Osnovna sintaksa sistemske specifikacije.

Grafične predstavitve so potrebne v načrtih delovanja sistema, medtem ko so besedne potrebne v SDL-programih. Med obema vrstama predstavitve je enolična preslikava. Če imamo na primer grafično predstavitev, lahko takoj zapišemo SDL-program.

Podobno kot deklariramo v računalniških programih kakršnekoli spremenljivke, moramo v SDL specificirati tudi signale. Meta-izraz za to je

signal<signal name>[(sort name{,<sort name>}*)]
{,<signal name>[(<sort name>{,<sort name>}*)]}*;

Temu ustreza primer

```
signal
Display1, Display2, Zastavica(Boolean);
```

Blok v sistemu predstavljamo s procesom, ki mu je lasten, in s povezavami, s katerimi komunicira z drugimi bloki v sistemu in/ali okoljem. Primer zapisa za blok Y v sistemu X je tako

```
system X;
channel c1 .....;
channel c2 .....;
block Y;
    signalroute r1 ......;
    signalroute r2 ......;
    process ProZ referenced;
    ...
endblock Blk_Y;
...
endsystem X;
```

Vidimo, da je blok v sistem vpet s kanali `channel` in v notranjosti bloka s potmi signalov `signalroute`.

Kanali v SDL so lahko enosmerni ali dvosmerni. Dva enosmerna sta ekvivalentna dvosmernemu kanalu. Vsaka smer v splošnem ni vezana samo na en signal, temveč na seznam signalov, ki se lahko pretakajo po kanalu. Za kanal velja naslednja meta-specifikacija

channel <channel name>
    from {<block name> | env}
        to {<block name> | env}
            with <signal list>;
    [from {<block name> | env}

```
        to {<block name> | env}
            with <signal list>;]
endchannel [<channel name>];
```

env pomeni okolje (environment). Praktična uporaba tega izraza v primeru kanala Inp_C je, recimo:

```
channel Inp_C
    from Dialog to env
        with Prikaz1, Prikaz2, Plačilo, Prazno;
    from env to Dialog
        with Tipka, Popravek;
end channel Inp_C;
```

Podobno specifikacijo imamo za pot signala, le da gre v tem primeru za proces in ne blok (proces je v notranjosti bloka)

```
signalroute <signal route name>
    from {<process name> | env}
        to {<process name> | env}
            with <signal list>;
    [from {<process name> | env}
        to {<process name> | env}
            with <signal list>;]
```

Pri izhodu iz bloka, se dogodi, da prideta skupaj kanal in pot signalov. Da je prehodnost signala omogočena, imamo povezavo

```
connect <channel name>
and <signalroute name>{, <signalroute name>}*;
```

Seveda mora biti seznam signalov enak tako pri poti kot kanalu. Kanal mora biti povezan na najmanj eno pot signalov, medtem ko je ena pot sigalov vezana na največ en kanal.

Na kratko omenimo še sintaksne elemente procesa. Proces je vedno v določenem bloku. Lahko je specificiran lokalno, lahko pa tudi oddaljeno. V primeru oddaljene specifikacije govorimo o referenciranem procesu, katerega besedno opišemo s stavkom

```
process <process name> referenced;
```

Vsak proces ima svojo deklaracijo in telo. Telo procesa podaja vedenje procesa. Besedna predstavitev obeh delov oziroma procesa kot celota, je

```
process <process name> referenced;
```

```
        <process declarations>
        <process body>
    endprocess [<process name>];
```

Poglejmo si telo procesa nekoliko detajlnejše.

11.2.3 Stanja, vhodi in prenosi

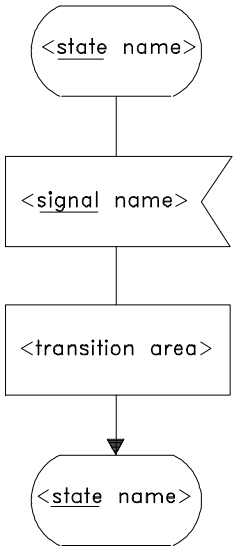
Telo procesa oziroma vedenje le-tega je postavljeno v okvir avomata stanj (state-transition machine). V takšnem avtomatu imamo samo dve možnosti delovanja. Ali ta avtomat stoji v določenem stanju (stacionarno stanje) ali pa prehaja med dvemi stanji. Seveda se v času nahajanja v stabilnem stanju, v sistemu, v katerega sodi proces, marsikaj dogaja. Čas nahajanja v stabilnem stanju lahko vzamemo kot časovno kontrolo dogajanj, ki so pogojena z aktivnim stabilnim stanjem avtomata. Avtomat stanj je podan s trojko $A = \{X, Y, \delta\}$. Spoznali smo že, da je X množica vhodov, Y množica stanj in δ relacija med množicama X in Y. Značilnost avtomata A v SDL procesu je, da se vhode $x \in X$ jemlje iz vhodne vrste procesa. Besedna predstavitev funkcije prehajanja stanj $\mathrm{D}^1 q = \delta(x, y)$ je naslednja

```
state <state name>;
    input <signal name>;
        [<transition>]
    nextstate <state name>;
[endstate [<state name>];]
```

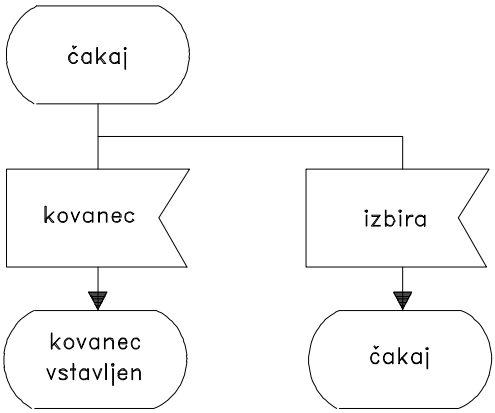
Grafično predstavitev tega prehajanja stanj vidimo na sliki 11.3. Vidimo, kako označujemo stanja, prehode in vhode. Vhod je označen v smislu puščice vstopnega signala. Gornje stanje je sedanje in spodnje naslednje stanje. Na sliki 11.4 vidimo primer dela avtomata stanj, ki se nanaša na vnašanje kovanca v napravo, ki izdaja znamke. Prikazani sta stanji `čakaj` in `kovanec_vstavljen`. Če se naprava znajde v stanju `kovanec_vstavljen`, pomeni, da se že odpira okence in ob tem ponudi znamko, tistemu, ki je vstavil kovanec. Lahko pa se pred tem opravi v napravi še marsikaj drugega, kar pa ni specificirano v opazovanem bloku. V kolikor do vhodnega signala `kovanec` še ni prišlo, se proces nahaja še naprej v stanju `čakaj.`

11.2.4 Opravilo

V SDL je opravilo (task) prirejanje nove vrednosti znani spremenljivki. Kot pri procesu vedenje procesa podaja telo procesa, navežemo vedenje prire-



Slika 11.3: Grafična predstavitev prehajanja stanj procesa.



Slika 11.4: Prikaz procesa na osnovi avtomata stanj.

janja nove vrednosti spremenljivki tudi na ustrezno telo. Tako je opravilo neke vrste proces. Ker gre za matematično prirejanje vrednosti spremenljivki, kar je zelo pogostna operacija pri izvajanju programa, je dobro, da je opravilo posebej izraženo in ga ne mešamo z ostalimi aplikativnimi procesi v sistemu. Besedna predstavitev opravila je

```
state <state name>;
    input <signal name>
        ...
        task <task body>
        ...
    nextstate <state name>
[endstate [<state name>];]
<task body>::=
    <variable name>:=<expression>
        {,<variable name>:=<expression>}*
    <informal text>{,<informal text>}*
```

<informal text> pomeni, da gre za besedno spremenljivko (string), ki jo lahko procesiramo, podobno kot imamo to tudi v izvajalnih programskih jezikih. Gornji besedni predstavitvi ustreza grafična predstavitev na sliki 11.5.

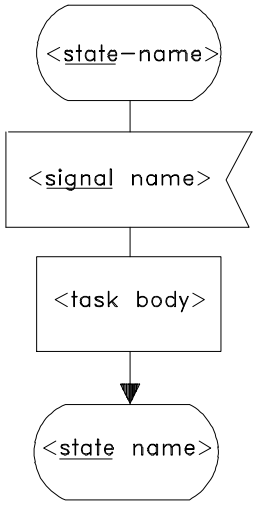
Na sliki 11.6 vidimo uporabo gornje opredelitve opravila. Glede na sliko 11.4 imamo pri opravilu vnešene prireditve vrednosti `x:=value(kovx)` oziroma `x:=0`. Preden pridemo v stanje `kovanec_vstavljen`, se izvrši `x:=value(kovx).` S tem naprava za izdajanje znamk spozna, kako velik kovanec je bil vstavljen v napravo. Za večji kovanec izda več znamk ali bolj vredno znamko. Ta izbira je možna, ker ima spremenljivka x ustrezno vrednost.

Da prirejanje vrednosti spremenljivki po sliki 11.6 deluje, mora biti predhodno napisana deklaracija

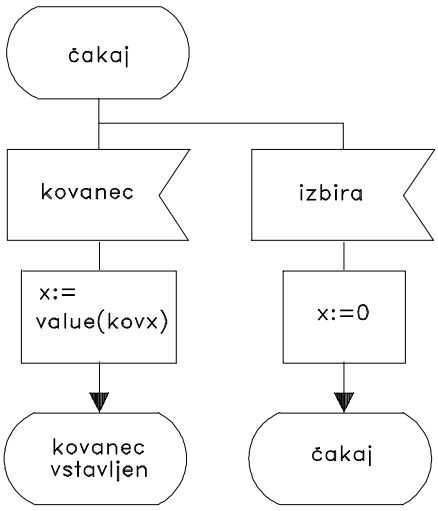
```
dcl kovx Integer;
dcl x Integer
```

11.2.5 Vejanje prehajanja stanj

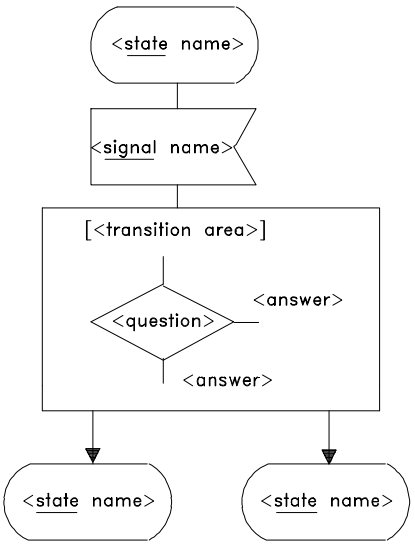
Pri potekanju procesa v računalniškem sistemu prihaja do vejanja, ki ga povzročajo določeni logični pogoji. V SDL se to vejanje izvaja v področju prehajanja stanj, ki je lahko posebej specificirano. Po vhodnem signalu



Slika 11.5: Grafična predstavitev opravila.



Slika 11.6: Primer postavitve opravil.

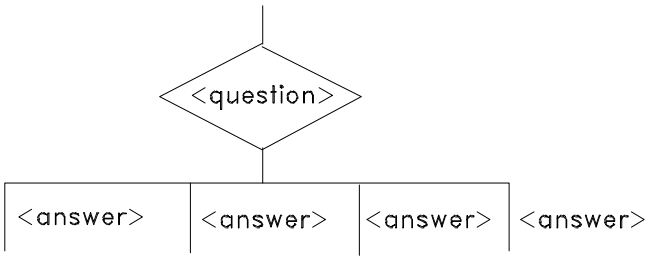


Slika 11.7: Splošna grafična predstavitev vejanja v SDL.

pride do prehoda stanja a v stanje b ali v stanje c, zavisno od logičnega pogoja. Meta-zapis za ta dogodek je naslednji

```
state <state name>;
    input <signal name>;
        ...
        decision <question>
            <answer>:<transition branch>
                nextstate <state name>
            <answer>:<transition branch>
                nextstate <state name>
            ...
        enddecision;
        ...
    [endstate[<state name>];]
    <answer>:=(<range>|<informal text>)|else
```

Na sliki 11.7 vidimo grafični ekvivalent k zadnjemu besednemu zapisu vejanja. V izvajalnih programskih jezikih smo vajeni, da je odgovor decizije logična vrednost 0 ali 1 ('false', 'true'). V SDL so lahko različni odgovori, na



Slika 11.8: Vejanje z več odgovori.

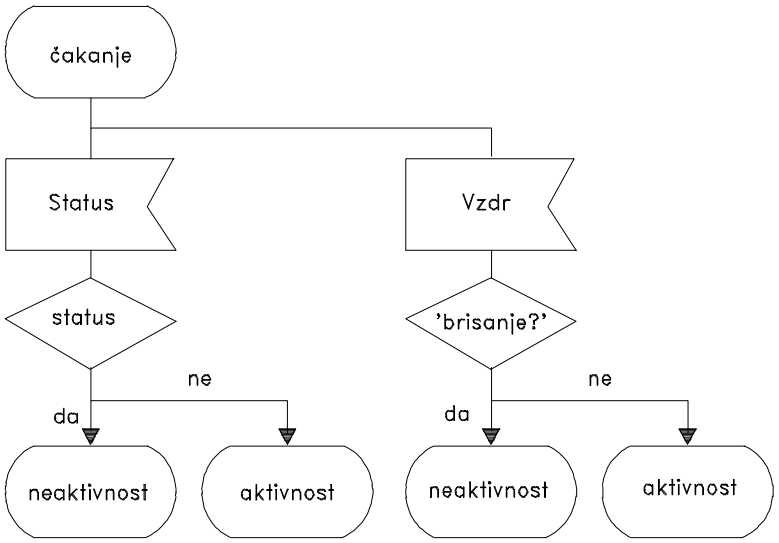
primer <1 ali 2 ali >7 itd. na vprašanje $x+y*2$. V odgovoru lahko srečamo celo `else`, ki pokriva vse ostale vrednosti. Če pride do odgovora 'false' za vprašanje $x+y*2$, pomeni, da je prišlo do napake. Zadnje pomeni, da je pri odločanju prišlo do področja, ki ni specificirano. Uporabniki morajo zato paziti, da takšna nespecificirana področja pri odločanju ne nastopijo sicer pride do (dinamične) napake. Na primer deljenja z 0 se je potrebno izogniti, ker pride do omenjene napake, ki zahteva ponovni zagon procesiranja.

Vsako vejanje mora imeti vsaj dva odgovora. Kako rišemo vejanje z več odgovori, vidimo na sliki 11.8. Jasno je, da med odgovori ne sme biti nobenega preseka. Samo en odgovor je lahko 'else'-odgovor, ki je dopolnilo k vsem ostalim odgovorom. Vprašanja in pogoji vejanja morajo biti iste vrste, lahko pa so tudi neformalno besedilo, kot to vidimo na sliki 11.9. Primer obravnava pri stanju `čakanje` skok v aktivnost, če pride do signala `Status` in skok v vzdrževanje, če pride do signala `Vzdr`.

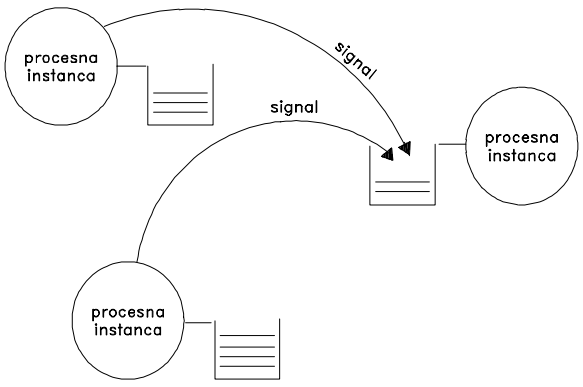
11.2.6 Komuniciranje med procesi

Kot smo že spoznali (slika 11.1), ima vsaka procesna instanca (kot stanje osnovnega diagrama prehajanja stanj) svojo FIFO-vrsto, v kateri čakajo signali, da po aktiviranju potujejo tja, kamor so namenjeni. Na sliki 11.10 vidimo 3 procesne instance, pri katerih vsaka zase pošilja svoj signal v tretjo procesno instanco, kjer se uvrstita v vhodno vrsto.

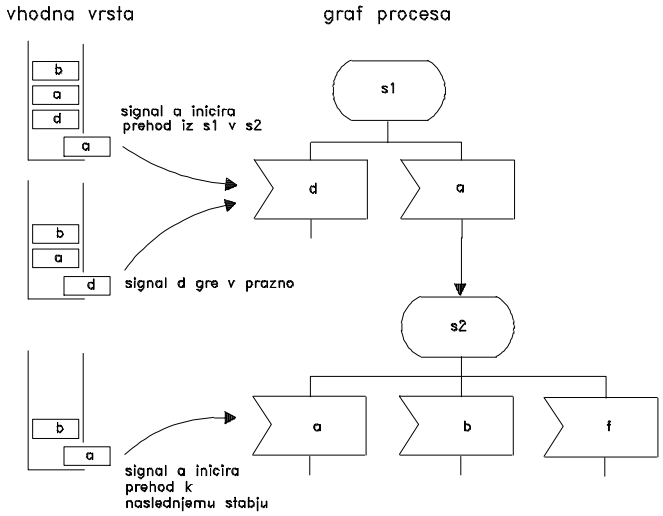
Kako signali, ki so v vhodni vrsti, vplivajo na potek procesa, vidimo na sliki 11.11. V gornji vrsti na omenjeni sliki imamo signale `a`, `b`, `d`, in `b`. Ker je vrsta tipa FIFO, je najprej na vrsti signal `a`, ki zahteva prehod iz stanja `s1` v stanje `s2`, kot je to določeno v diagramu procesa. Torej se usmerimo pri `a` v smeri stanja `s2`. Naslednji signal, to je `d` (glej drugi primer vhodne



Slika 11.9: Navadno vejanje in vejanje pri neformalnem besedilu.



Slika 11.10: Princip komuniciranja med procesnimi instancami.



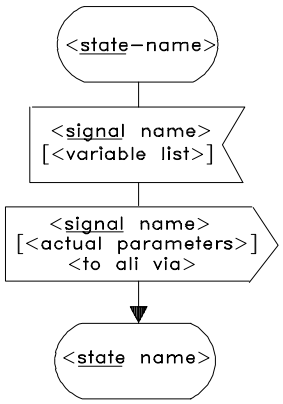
Slika 11.11: Vpliv vhodne vrste na SDL diagram poteka.

vrste), zahteva spremembo stanja `s1` v neko stanje, ki ni `s2`, vendar nas je signal `a` že usmeril v stanje `s2`. Naslednji signal `a`, ki prihaja za `d`, že zahteva, da se stanje `s2` spremeni v stanje, ki v diagramu sledi vhodu `a` itd.

Do sedaj smo opazovali le vstop signala v diagram poteka procesa, ki se je moral nekje pojaviti kot output. Kako se generira signal v diagramu SDL-procesa? Pri prehodu med dvemi stanji lahko generiramo signal, in tudi postavimo njegove parametre, z namenom, da ga pošljemo iz opazovanega procesa v drugi proces. Na sliki 11.12 vidimo grafično predstavitev tegá. Vhod je razširjen za spremenljivke, ki morajo biti iste sorte, kot vrednosti v okviru izhoda. Korespodenca med vrednostmi izhoda in spremenljivkami vhoda je napravljena na osnovi vrstnega reda v seznamu. Možne so opustitve nekaterih vrednosti izhoda ali pa spremenljivk vhoda. Naj navedemo primer za takšne opustitve: izhod, `output X(1,2,,0,)` in vhod, `input X(i,,j,0)`. V tem primeru je vrednost 2 v signalu izgubljena ter spremenljivka `j` nedefinirana.

11.2.7 Naslavljanje

Vsak SDL-proces ima naslov enote. Posebnost je v tem, da naslov ne postavlja uporabnik, ampak se generira samodejno na osnovi SDL-sistema



Slika 11.12: Postavitev signala v SDL-procesu.

(SDL machine). S tem je garantirano, da ima prav vsak proces pravilno postavljen naslov. Naslov procesa ni navadno ime procesa, temveč v SDL za to obstajajo posebni predefinirani izrazi (sort *Pld*). Vsak proces ima štiri predefinirane *Pld*-izraze:

- `self`, postavljanje naslova samega procesa
- `sender`, postavljanje naslova s pošiljanem na koncu signala
- `offspring`, naslov nanovo kreiranega procesa
- `parent`, naslov procesa, ki je generiral trenutni proces.

Pred kreiranjem sistema imajo vsi omenjeni izrazi vrednost 0.

Poznamo dve vrsti naslavljanja: implicitno in eksplicitno naslavljanje. Eksplicitno naslavljanje uporablja konstrukt `to`. Ta se pojavlja v izhodu, kot na primer: `A to sender` ali `B to offspring` itd. Implicitno naslavljanje uporablja konstrukt `via`. Tako imamo za implicitno naslavljanje izhod, kot na primer: `A via Sr1`, pri čemer je `Sr1` signal, ki prehaja iz enega procesa v drugega.

11.3 INTEGRACIJE ORODIJ IN OKOLIJ

V okviru Esprit projekta ATMOSPHERE (projekt št. 2565, [23]) je bilo zastavljeno in reševano vprašanje, kaj je CBSE (Computer Based System

Engineering)? V okviru tega so strokovnjaki začeli razčiščevati, kakšne so skupne in kakšne različne značilnosti raznih področij dela, kot so programersko inženirstvo, VLSI inženirstvo, komunikacijsko inženirstvo itd. Med drugim so spoznali, kako pomembna je integracija metod in okolij za načrtovanje, opisovanje ter specificiranje računalniško-komunikacijskih sistemov. V tem razdelku želimo le prikazati razsežnosti problema omenjenega inženirstva.

V prejšnjih razdelkih tega poglavja smo spoznali le osnovni del SDL, ki sicer zavzema vidno mesto pri določanju večjih komunikacijko-računalniških sistemov. Računalnikarji, komunikacisti in telekomunikacisti, skratka informacijski inženirji, morajo računati na hitro se bližajoči čas velikih integracij na področju že tako velikih računalniško-informacijskih sistemov, ki bodo ali so že v začetku načrtovani vsak zase na osnovi različnih vendar zelo znanih orodij. Integracije sistemov zahtevajo prehajanje iz enega orodja na drugega, vse s ciljem načrtovanja še večjih sistemov. Pomembna je tudi komponenta načrtovanja ne večjih, temveč boljših, bolj zanesljivih, učinkovitejših sistemov. Kot vidimo, je pomen integracije orodij in okolij in metod v njih pomembna postavka bodočega razvoja, ki nas vse bolj potiska v informacijsko družbo, informacijsko pogojeno poslovnost itd.

Za opisovanje, specifikacijo in načrtovanje strukturno organiziranih sistemov (med te zanesljivo sodijo računalniško-komunikacijski sistemi) poznamo vrsto splošno znanih okolij in vrsto firmnih okolij, ki so nastala ob načrtovanju določenih velikih sistemov. Med prve sodijo:

- ▷ SA (Structured Analysis, splošno orodje v UK),
- ▷ IM (Information Modeling, Entity-Relationship Diagrams, razširjeno orodje, ki je izšlo iz orodja za postavljanje baz podatkov),
- ▷ VHDL (VHSIC Hardware Description Language; Very High Speed Integrated Circuits, DoD, pomen za IEEE standardizacijo),
- ▷ HOOD (Hierarchical Object-Oriented Design, European Space Agency), SDL (Specification and Desription Language, CCITT) in
- ▷ LOTOS (Language Of Temporal Ordering Specification, ISO OSI),

med druge pa sodijo:

- ▷ Π (PEACOCK, Esprit 1 project),

- ▷ HIT (računalniško orodje kot osnova za modeliranje performans sistema),
- ▷ AKL ("AufgabenKLärung" orodje firme Siemens) in
- ▷ COLD (Common Object-oriented Language for Design, Philips Research).

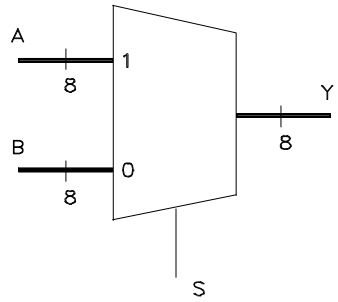
V teh okoljih najdemo konceptualne, načrtovalske (design), postavljalne in produkcijske modele sistema. Eni modeli so boljše postavljeni v enih in drugi v drugih okoljih. Načrtovalska sfera velikih računalniških in podobnih sistemov poskuša ugotoviti, za katero zvrst oziroma tehniko je katero okolje oziroma orodje boljše (bolj učinkovito) in za katero slabše (manj učinkovito). V kolikor vzamemo področje tehnike določanja zahtev sistema, področje načrtovalske tehnike in področje realizacijske tehnike, je razvrstitev gornjih okolij oziroma orodij naslednja:

- *Tehnika določanja zahtev*: SA, IM, AKL
- *Tehnika načrtovanja*: HOOD, COLD, SDL, HIT, Π
- *Tehnika realizacije*: neposredno nobeno okolje oziroma orodje. Še najbližje temu je VHDL in nekoliko manj HIT.

Predvsem evropski strokovnjaki (EU) delajo na različnih integracijah, da bi dosegli čim boljša okolja za specifikacijo, načrtovanje in postavljanje velikih in kvalitetnih računalniško-komunikacijskih sistemov. Potekajo obsežne študijske razprave o naslednjih integracijah oziroma podobnostih:

SA/IM-HOOD
AKL-SDL
LOTUS-SDL
COLD-SDL
Π-HIT
SDL-HIT
VHDL-HIT
SDL-VHDL.

Podčrtani SDL v zadnjem spisku nazorno prikazuje, da igra pri integraciji SDL osrednjo vlogo v prizadevanjih za izboljšavo okolij za opisovanje, specifikacijo in načrtovanje RSS. Prav zato smo ga v tem poglavju, vsaj v fragmentih, predstavili.



Slika 11.13: Postavke 8-bitnega multipleksorja z enostavno selekcijo.

11.4 VHDL

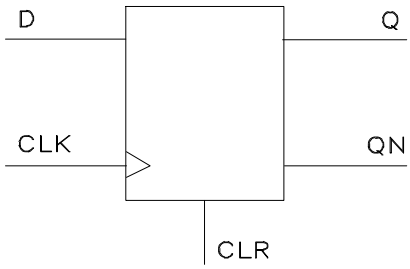
Kot se odlikuje SDL po svoji nedvomnosti opisovanja, specificikacij in načrtovanja sistema, lahko isto trdimo za VHDL. Prav zaradi te značinosti, se je v zadnjih desetih letih VHDL močno uveljavil. Temeljne ključne besede tega jezika so: vedenje, časovnost in strukturalnost RSS. Vsaka od teh besed vodi v svoj model tako, da VHDL pozna tri vrste modelov: model vedenja (reakcija na vhodne podatke, vmesni rezultati, izhodni, končni rezultati), časovni model (opredelitev časa k vsemu, kar se dogodi pod vedenjem) in strukturalni model (hirarhične dekompozicije sistema). Poleg teh modelov je še model podatkov. Vsaki postavki v VHDL (konstante, signali, spremenljivke) pripada določen tip, za tega pa je značilna možna množica vrednosti, ki se lahko dogodijo pri potekanju procesa.

VHDL ima, kot drugi jeziki, svojo sintakso in semantiko. V okviru sintakse se vse dogaja v skladu z BNF (Backus-Naur Form), kar smo zapisali tudi za SDL. V okviru semantike imamo pravila, ki omogočaja razpoznavaje, da imajo napisane besede pomen stavka, ko gre za informacije o RSS.

Na kako visokem nivoju podajamo v VHDL podatke, si poglejmo na primeru podajanja MX kot decizijske enote, D-celice kot pomnilnega elementa in Mooreovega avtomata.

Na sliki 11.13 vidimo značilne postavke za 8-bitni multipleksor, ki pri `S` $= 1$ prepušča na izhod spremenljivko `A` in pri `S` $= 0$ spremenljivko `B`. Računamo, da je zakasnitev od vhoda do izhoda `TDP` $= 1$ ns. VHDL-program, ki podaja omenjeni multipleksor, je

```
entity Mux8 is
```



Slika 11.14: Postavke za sinhronsko D-pomnilno celico.

```
    generic (TDP : TIME := 1 ns);
    port (A, B : in BIT_Vector (7 downto 0);
    Sel :  in BIT := '1'; Y : out BIT_Vector (7 downto 0));
end;
architecture Behave of Mux8 is
begin
    Y <= A after TPD vhen Sel = '1' else B after TPD;
end;
```

<= ni relacija, manjše ali enako, temveč prireditvena puščica. Vidimo, da je program takorekoč podan v stavkih, kar vse skupaj zelo približa angleškemu naravnemu jeziku. Glede na to nam ni potrebno dati praktično nobenega komentarja, če le hkrati opazujemo sliko za MX in program. `generic` je nekaj podobnega kot `port`, samo da se nanaša na konstante, ki delujejo nad opazovano entiteto. Nasprotno se torej `port` nanaša na spremenljivke.

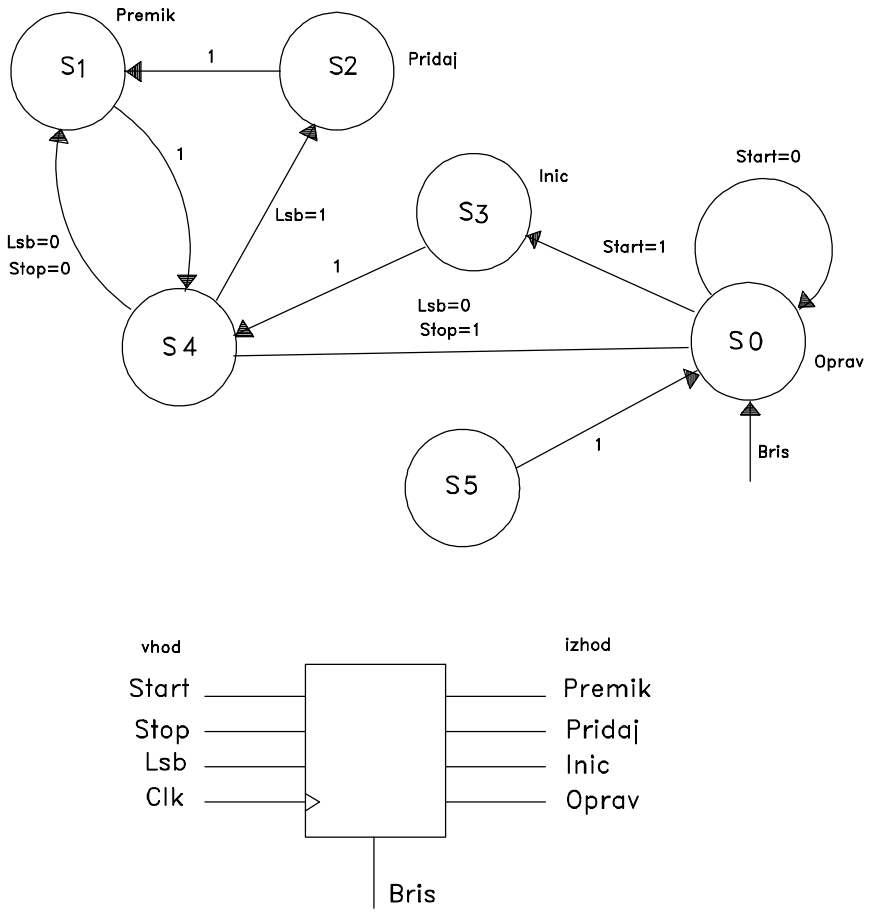
Na sliki 11.14 so podane spremenljivke sinhronske D-pomnilne celice z brisanjem, ki je asinhronsko. Zakasnilni čas, ki je dopusten od signala `CLR` do izhoda `Q` oziroma `QN`, je 2 ns. Ravno takšno zakasnitev imamo med `CLK` in `Q` oziroma `QN`. Program, ki pokriva takšno pomnilno celico, je

```
entity DFF is
    generic (TQR : TIME := 2 ns; TCQ := 2 ns);
    port (CLR, CLK, D : in BIT; Q, QB : out BIT);
end;
architecture Behave of DFF is
signal Qi :  BIT;
begin QB <= not Qi; Q <= Qi;
```

```
process (CLR, CLK) begin
    if (CLR = '1' then Qi <= '0' after TRQ;
        elsif CLK'EVENT and CLK = '1'
    then Qi <= D after TCQ;
    end if;
end process;
end;
```

S področja računalniške aritmetike si poglejmo še Mooreov avtomat na sliki 11.15. Kot prikazuje bločna shema, na dani sliki spodaj, imamo štiri vhodne črke, štiri izhodne črke in pa šest stanj. Avtomat lahko brišemo s signalom `Bris`. Stanje `S0` je začetno stanje, v katerem se prične delovanje avtomata ('`ready`'). V tem stanju se tudi znajdemo, ko je vse delo opravljeno; tedaj dobimo izhodni signal (črko) `Oprav`. Ko nastopi `Start=1`, se prične inicializacija, ki jo zahteva izhodni signal `Inic` v stanju `S3`. Ko je inicializacija končana, gremo v stanje `S4` ne glede na to, kakšne so vhodne črke (zato je prehod določen z 1). V stanju se testira akumulator, če je spodnji del le-tega prazen. Če je prazen, je `Lbs=1`, kar nas vodi v stanje `S2`. To stanje z izhodno črko `Pridaj` zahteva v opazovani aritmetični enoti delna vsoto. Ko je ta končana, gremo v avtomatičen premik vsebine, kar omogoča množenje. Izhodna črka, ki to zahteva, je `Premik`. Natančen opis delovanja podanega avtomata podaja VHDL-program

```
entity Mavt is
    generic (TPD : TIME :* 1 ns);
    port (Start, Clk, Lsb, Stop, Bris:  in BIT;
    Init, Premik, Pridaj, Oprav :  out BIT;
end;
architecture Moore of Mavt is
type STATETYPE is (S0, S1, S2, S3, S4, S5);
signal State:  STATETYPE;
begin
Inic <= '1' after TPD when State = S3
    else '0' after TPD;
Pridaj <= '1' after TPD when State = S2
    else '0' after TPD;
Oprav <= '1' after TPD when State = S0
    else '0' after TPD;
Premik <= '1' after TPD when State = S1
    else <= '0' after TPD;
```



Slika 11.15: Mooreov avtomat za postavitev delne vsote in njega premika pri množenju, podan z diagramom prehajanja stanj.

```
Process (CLK, Bris) begin
    if Bris = '1' then State <= S0
    elsif CLK'EVENT and Clk = '1' then
        case State is
        when S3 => State <= S4;
        when S4 =>
            if Lsb => '1' then State <= S2;
            elsif Stop => '0' then State <= S1;
            else State <= S0;
            end if;
        when S2 => State <= S1;
        when S1 => State<= S4;
        when S0 =>
            if Start = '1' then State <= S3;
        end case;
    end if;
end process;
end;
```

Na osnovi gornjih treh primerov se da zaključiti, kako enostavno lahko podajamo najrazličnejša logična delovanja RSS. Vidimo tudi, kako strog je VHDL do zakasnitev, ki nastajajo kot asinhronski priveski v delovanju logičnih shem. Zakaj je to potrebno, smo že govorili v okviru PPAL- gradnikov.

VHDL slovi po tem, da ima bogate knjižnice in programske pakete. Med te osnovne pripomočke sodijo:

- načrtovalska knjižnica (VHDL design library, LRM11.2)
- paket podprogarmov, procedur, funkcij (VHDL LRM2.5-2.6)
- standardni paket za čas, real in integer tipnih postavk (IEEE-stadard, VHDL LRM14.2)
- paket Std_logic_1164 (IEEE-standard 1164-1993) za tip dvojiške postavke BIT
- paket TEXTIO za vhodno-izhodna besedila (del STD-knjižnice, VHDL LRM14.3)
- paketi za aritmetiko (IEEE-standard 1164)

- matematični paketi (IEEE 1076.2,1996)
- paketi za komponente
- paket za ASIC-sinteze in
- drugi.

LRM pomeni VHDL Language Reference Manual. Poleg omenjenega IEEE standarda VHDL pokriva tudi standard MIL-STD-454 (DoD, ZDA) kot medij za dokumentiranje, simulacije in verifikacije ASIC-shem.

11.5 TRANSFORMACIJA SDL-VHDL

Pulkkinen in Kronlöf sta v [23] podala prehod iz SDL na VHDL in narobe, ob tem pa sta tudi podala potrebo po takšnem prehajanju iz okolja na okolje. Ker se tako v SDL- kot tudi v VHDL-sintaksi uporablja Backus-Naur Form koncept, je vrsta stvari primerljivih. Seveda pa ostane tudi nekaj stvari, ki so težje transparentno zložljive, vendar pa je z daljšim opisovanjem možno tudi to premostiti. Transformacijo SDL-VHDL poskušajo doseči predvsem na osnovi sintaktičnih kategorij v obeh okoljih, v okviru katerih lahko govorimo o ekvivalentnosti. Za primerjavo, do kakšne mere sta si SDL in VHDL načelno podobna, si poglejmo le <blok definition> v SDL in <blok definition> v VHDL, za katera lahko rečemo, da sta ekvivalentna.

SDL-blok:

```
<block definition>::=
    BLOCK <block name>;
        {<data definition>}*
        {<block behaviour definition>|
       <textual block substructure reference>}
        {<data definition>}*
    ENDBLOCK <block name>
<block behaviour definition>::=
    {<signal route definition> <chanel to route
connection>}*
    <textual process reference>
    {<signal route definition> <channel to route
connection>*
```

```
<signal route definition>::=
    SIGNALROUTE <signal route name>
        {FROM <process name> TO ENV | FROM ENV TO
<process name>}
        WITH <signal name>;
<channel to route connection>::=
    CONNECT <channel name> AND <signal route name>;
<block substructure definition>::=
    SUBSTRUCTURE <block substructure name>;
        {<textual block reference>|<channel definition>|
            |<channel connection>|<signal definition>|
            |<data definition>}+
    ENDSUBSTRUCTURE <block substructure name>;
<channel connection>::=
    CONNECT <channel name> AND {<subchannel name>;}+
```

VHDL-blok:

```
<block definition>::=
    <block interface definition>
    {<block behaviour definition>|
    <block substructure definition>}
<block interface definition>::=
    {<use clause>}*
    ENTITY <bloch name> IS
        PORT(<port list>);
    END <block name>;
<block behaviour definition>::=
    ARCHITECTURE behaviour OF <block name> IS
        {SIGNAL <signal name>:
      <signal type> [::= <expression>];}*
    BEGIN
        <process definition>
    END behavioural
<block substructure definition>::=
    ARCHITECTURE structural OF <block name> IS
        {SIGNAL <signal name>:<signal type>;)+
        {COMPONENT <block name>
            PORT (<port list>);
        END COMPONENT;)+
```

```
    BEGIN
        {<instance name> :  <block name>
            PORT MAP (<port map list>) ;)+
    END structural;
<signal type>::=
    <sort name>
```

Že ob površnem znanju o Backus-Naur notaciji je možno na gornjem primeru <block definition> videti, da je v postavljenih definicijah toliko podobnosti, da si človek težko predstavlja, da sta obe okolji nastajali neodvisno en od drugegega.

Problem sodobnega načrtovalca RSS je v tem, da, če je sistem podan v okolju SDL (VHDL), ga zna interpretirati tudi v VHDL (SDL). Zadnje je (predvsem pa bo v bodočnosti) potrebno, če poskuša dva neodvisna podsistema sestaviti (integrirati) v sistem in je en podan s SDL, drug pa z VHDL. Preko SDL v velik sistem prihajajo komunikacijski in preko VHDL računalniški sestavi. Zadnje je pomembna simbioza, ki s tehnološkega vidika odpira pot v velike informacijske sisteme, ki delujejo na osnovi porazdeljenega procesiranja, porazdeljenjega pomnjenja in porazdeljene kontrole. Povezave med enotami zahtevajo protokolno tehniko, ki si jo težko predstavljamo brez SDL in podobnih orodij.

Del V

DODATEK

Dodatek A

PROGRAMI

A.1 ZAKAJ IN KAKO SO POSTAVLJENI PROGRAMI?

V Dodatku podajamo MATLAB-programe `X.M`, ki so podani v nekaterih poglavjih za boljšo razlago podanih postopkov in metod. Vrstica s simbolom % na svojem začetku pomeni komentar, tako da posebne razlage izven samih programov nam ni potrebno delati. V programski vrstici, ki pričenja s '`%Functions: ....`' so zapisane funkcije, ki jih potrebujejo programi in jih moramo izdelati sami na osnovi MATLAB-instrukcij. Programi za takšne funkcije so podani pod posebnimi naslovi. Imena funkcij so `Y.M`, torej so funkcije samostojni MATLAB-programi, ki jih lahko uporabimo kjerkoli, ne samo na podanih, predvidenih mestih.

Namen podanih programov in funkcij je med drugim tudi v tem, da bi bralec sam posegel po okolju MATLAB. Z njim bi si postavil za iztočnico, v tem delu naveden, program z ustreznimi funkcijami, za tem pa bi začel sam eksperimentirati s podatki, spreminjati programe, spreminjati in dopolnjevati programirani postopek itd.

Podani programi so eksperimentalni in kot taki lahko neoptimalni. V takšni obliki bolj ustrezajo namenu učbenika kot sicer.

A.2 Program 1. PARTITION PROCESS OF LOGICAL SHEME

```
%Optimization of partition process
%File name: LPR.M
```

```
%Created: V., LRSS, January 1998
%Functions: linpro(x)
%
%index-identification of variables
%x=[x12 x13 x14 x15 x16 x23 x24 x25 x26 x34 x35 x36 x45 x46
%i   1   2   3   4   5   6   7   8   9  10   11  12  13  14
%x56]
% 15
%initial vector
x0=[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0];
x=constr('linpro',x0);
[f,g]=linpro(x);
dolx=length(x);
for i=1:dolx
   if x(i)<0.5
      x1(i)=0;
   else
      x1(i)=1;
   end;
end;
%results
x
%go to Boolean variables
x1
%optimal value of function
f
%value of function at Boolean variables
f1=x1(1)+x1(2)+x1(3)+x1(4)+x1(5)+x1(6)+x1(7)+x1(8)+x1(9)+...
 \ \ x1(10)+x1(11)+x1(12)+x1(13)+x1(14)+x1(15)
%restrictions, see linpro(x)
g
```

A.2.1 Funkcija 1: Make optimal partition process

```
%File name: LINPRO.M
%Created: V., LRSS, January 1998
%Main program: LPR.M
%
function [f,g]=linpro(x)
```

```
f=x(1)+x(2)+x(3)+x(4)+x(5)+x(6)+x(7)+x(8)+x(9)+x(10)+x(11)+...
  x(12)+x(13)+x(14)+x(15);
g(1,1)=5-x(1)-x(2)-x(3)-x(4)-x(5)-4;
g(2,1)=5-x(1)-x(6)-x(7)-x(8)-x(9)-4;
g(3,1)=5-x(2)-x(6)-x(10)-x(11)-x(12)-4;
g(4,1)=5-x(3)-x(7)-x(10)-x(13)-x(14)-4;
g(5,1)=5-x(4)-x(8)-x(11)-x(13)-x(15)-4;
g(6,1)=5-x(5)-x(9)-x(12)-x(14)-x(15)-4;
g(7,1)=2*x(1)+x(2)+x(3)+x(4)-5;
g(8,1)=2*x(1)+x(6)+2*x(7)+x(9)-5;
g(9,1)=x(2)+x(6)+x(10)+x(11)+x(12)-5;
g(10,1)=x(3)+2*x(7)+x(10)+2*x(13)+2*x(14)-5;
g(11,1)=x(4)+x(8)+x(11)+2*x(13)+x(15)-5;
g(12,1)=x(9)+x(12)+2*x(14)+x(15)-5;
%Index interpretation of variables
%x=[x12 x13 x14 x15 x16 x23 x24 x25 x26 x34 x35 x36 x45 x46
%i   1   2   3   4   5   6   7   8   9  10   11  12  13  14
%x56]
% 15
%restrictions
g(13,1)=2-x(1)-x(2)-x(6);
%g(13,1)=x(1)+x(2)+x(6);
g(14,1)=2-x(1)-x(7)-x(3);
%g(14,1)=x(1)+x(7)+x(3);
g(15,1)=2-x(1)-x(8)-x(4);
%g(15,1)=x(1)+x(8)+x(4);
%g(16,1)=2-x(1)-x(9)-x(5);
g(16,1)=x(1)+x(9)+x(5);
%g(17,1)=2-x(2)-x(3)-x(10);
g(17,1)=x(2)+x(3)+x(10);
g(18,1)=2-x(2)-x(4)-x(11);
%g(18,1)=x(2)+x(4)+x(11);
g(19,1)=2-x(2)-x(5)-x(12);
%g(19,1)=x(2)+x(5)+x(12);
g(20,1)=2-x(3)-x(4)-x(13);
%g(20,1)=x(3)+x(4)+x(13);
%g(21,1)=2-x(4)-x(5)-x(15);
g(21,1)=x(4)+x(5)+x(15);
%g(22,1)=2-x(6)-x(7)-x(10);
```

```
g(22,1)=x(6)+x(7)+x(10);
%g(23,1)=2-x(6)-x(8)-x(11);
g(23,1)=x(6)+x(8)+x(11);
g(24,1)=2-x(6)-x(9)-x(12);
%g(24,1)=x(6)+x(9)+x(12);
%g(25,1)=2-x(7)-x(8)-x(13);
g(25,1)=x(7)+x(8)+x(13);
%g(26,1)=2-x(7)-x(9)-x(14);
g(26,1)=x(7)+x(9)+x(14);
%g(27,1)=2-x(8)-x(9)-x(15);
g(27,1)=x(8)+x(9)+x(15);
%g(28,1)=2-x(10)-x(11)-x(13);
g(28,1)=x(10)+x(11)+x(13);
g(29,1)=2-x(10)-x(12)-x(14);
%g(29,1)=x(10)+x(12)+x(14);
%g(30,1)=2-x(13)-x(13)-x(15);
g(30,1)=x(13)+x(13)+x(15);
```

A.3 Program 2. MODULE PLACEMENT

```
%Lower bound on minimal connection length SMTDP
%File name: SMTDP.M
%Create: V.,LRSS,FRI,February 2,1998
%Functions: TESTA.M,ORDMAT.M,OREDERC.M,
%OPTIRES.M,NODIAG.M,NEKSTA.M,MUNK.M,
%MIOUT0.M,MINKOS.M
clear
'     Connection and Distance Data      '
c=[0 3 2 4 0; 3 0 4 0 0; 2 4 0 3 0; 4 0 3 0 0; 0 0 0 0 0]
d=[0 2 5 3 1; 2 0 2 2 3; 5 2 0 3 6; 3 2 3 0 1; 1 3 6 1 0]
global xset minix imin ste
len=length(c(1,:));
imin=1; %index of matrix minimum
ste=0; %number of elements in xset one step before
minix=zeros(size(1:len)); %min of matrices a...
xset=zeros(size(1:2*len)); %initial set of independent zeros
[va vb vc]=ordmat(c); %up-triangle of matrix c
[ud ve vf]=ordmat(d); %up-triangle of matrix d
'     Lower Bound on Minimal Connection Length: SMTDP     '
```

```
smtd=sum(vc.*ve)
nodiac=nodiag(c); %matrix c without diagonal
nodiad=nodiag(d); %matrix d without diagonal
%vector ordering for matrices c and d without diagonal
dolz=length(c(1,:));
for i=1:dolz
  ordc(i,:)=ordvek(nodiac(i,:));
end
nodid=nodiad;
for i=1:dolz
  [orddob(i,:),ordd(i,:)]=ordvek(nodid(i,:));
end
%SMTDP with actual placement element i on position (j)
for i=1:dolz
  for j=1:dolz
    amat(i,j)=sum(ordc(i,:).*ordd(j,:));
  end
end
'     Permutation Matrices, Minimal Costs Calculation       '
a=amat
%matrices going to minimum of cost function
[a1 a11]=munkr(amat)
[xset stev]=testa(a11);
a111=nexta(a11,stev)
[xset stev]=testa(a111);
a1111=nexta(a111,stev)
[xset stev]=testa(a1111);
a11111=nexta(a1111,stev)
'    Results with Minimal Costs: Permutations per Element     '
[ep st]=optires(a1111);
ep
'   Results with Minimal Costs: Number of SMTDP/Permutation   '
st
```

A.3.1 Funkcija 2. Test for row and column zeros

```
%File name: TESTA.M
%Created: V., LRSS, January 1998
%Main programs: LPR.M,MATT.M
```

```
function [xset,stev]=testa(at)
global xset ste
len=length(at(1,:));
stev=0;
xse=zeros(size(1:2*len));
nic=zeros(size(1:len));
for i=1:len
  if at(i,:)==nic;
    xse(i)=i;
  end
end
for j=1:len
  if at(:,j)==nic';
    xse(len+j)=j;
  end
end
xset=xset+xse;
end
for i=1:2*len
  if xset(i)>0
    stev=stev+1;
  end
end
```

A.3.2 Funkcija 3. Vector rotation of 180 degree

```
%File name: ORDVEK.M
%Create: V.,LRSS,FRI,February 1,1988
%Main program: LPR.M
function [ob,obv]=ordvek(v)
clear vekt;
clear obv;
dolz=length(v);
vekt=v;
for i=1:dolz
  ma=max(vekt);
  for j=1:dolz
    if vekt(j)==ma
      vekt(j)=0;
```

```
      ob(dolz-i+1)=ma;
      obv(i)=ma;
      break;
    else
    end
  end
end
```

A.3.3 Funkcija 4. Ordering of up-half matrix c

```
%File name: ORDMAT.M
%Create: V.,LRSS,January 1.Feb.1998
%Main program: LPR.M
function [re,r,obr]=ordmat(c)
dol=length(c(1,:));
clear rr
clear re
clear r
k=0;
for i=1:dol
  for j=1:dol
    if i<j
      k=k+1;
      re(k)=c(i,j);
    else
    end
  end
end
rr=re;
dolz=length(rr);
for i=1:dolz
  ma=max(rr);
  for j=1:dolz
    if rr(j)==ma
      rr(j)=0;
      r(dolz-i+1)=ma;
      obr(i)=ma;
      break;
    else
```

```
    end
  end
end
```

A.3.4 Funkcija 5. Number of permutations, permited for each element

```
%File name: OPTIRES.M
%Create: V.,LRSS,January 1.Feb.1998
%Main program: LPR.M
function [ep,st]=optires(ao)
leng=length(ao(1,:));
coun=0;
for i=1:leng
  for j=1:leng
    if ao(i,j)==0
      ep(i,j)=j;
    else
      ep(i,j)=0;
    end
  end
end
for i=1:leng
  stx=leng;
  for j=1:leng
    if ao(i,j)==0
      stx=stx-1;
    else
    end
  end
  stvl(i)=leng-stx;
end
st=prod(stvl);
```

A.3.5 Funkcija 6. Remove matrix diagonal

```
%File name: NODIAG.M
%Create: V.,LRSS, January 30,1998
%Main program: LPR.M
function cd=nodiag(c)
```

```
doli=length(c(1,:));
k=0;
h=0;
for i=1:doli
  for j=1:doli
    if (i<j)|(i>j)
      k=k+1;
      cd(i,k)=c(i,j);
    else
    end
    %if (i>j)
    %  h=h+1;
    %  cd(i,h)=c(i,j);
    %else
    %end
  end
  k=0;
  h=0;
end
```

A.3.6 Funkcija 7. Compute next matrix a

```
%File name: NEXTA.M
%Create: V.,LRSS, January 30,1998
%Main programs: LPR.M,MATT.M
function nex=nexta(an,stev)
global xset minix imin ste
dolz=length(an(1,:));
ste=ste+1;
%find break if matrix with all independent zeros
k=0;
for j=1:dolz
  for i=1:dolz
    if an(i,j)==0
      a=[i j];
      nicle(j+k,:)=a; %locations of all zeros in matrix an
    else
    end
    k=k+1;
```

```
  end
end
h=1;
kle=length(nicle(:,1));
for i=1:kle
  if min(nicle(i))>0
    nicl(h,:)=nicle(i,:);
    h=h+1;
  else
  end
end
an
nicl;  %matrix nicle without points [0, 0]
ledo=length(nicl(:,1));
bb1=1;
for i=1:ledo
  if nicl(i,1)==bb1
    bb1=bb1+1;
  end
end
bb1=bb1-1;
bb2=1;
for i=1:ledo
  if nicl(i,1)==bb2
    bb2=bb2+1;
  end
end
bb2=bb2-1;
coun=0;
for i=1:dolz
  for j=1:ledo
    if nicl(j,1)==i
      coun=coun+1;
    end
  end
  pono(i,1)=coun;
  coun=0;
end
coun=0;
```

```
for i=1:dolz
  for j=1:ledo
    if nicl(j,2)==i
      coun=coun+1;
    end
  end
  pono(i,2)=coun;
  coun=0;
end
pono;  %number of equal elements in matrix
bb3=0;
for i=1:dolz
  for j=1:dolz
    if i==j
    else
      if pono(i,1)==pono(j,1)
        if pono(i,2)==pono(j,2)
          bb3=1;
        end
      end
    end
  end
end
bb4=0;
for i=1:dolz
  for j=1:dolz
    if i==j
    else
      if pono(i,2)==pono(j,2)
        if pono(i,1)==pono(j,1)
          bb4=1;
        end
      end
    end
  end
end
bb1;  % 4 flags for solution of all independent zeros:
bb2;  % if bb1=bb2=5 and bb3=bb4=0 then matrix a... with all
bb3;  % independent zeros
```

```
bb4;
if bb1==dolz
  if bb2==dolz
    if bb3==0
      if bb4==0
        break;
      end
    end
  end
end
%do next matrix while not all independent zeros
if stev<dolz
  [mini,ani]=miout0(an);
    for i=1:dolz
      for j=1:dolz
        pog2000=ani(i,j)==2000;
        pogi1=xset(i)==i;
        pogj1=xset(dolz+j)==i;
        pogi2=xset(i)==j;
        pogj2=xset(dolz+j)==j;
        pog=(((pog2000*pogi1)*pogj1)*pogi2)*pogj2;
        if ani(i,j)<1000
          nexa(i,j)=ani(i,j);
        end
        if ani(i,j)==1000
          nexa(i,j)=mini;
        end
        if ani(i,j)==2000
          nexa(i,j)=2*mini;
        end
        mx=minix(imin-1);
        if pog==1
          nexa(i,j)=2*mini+mx;
        end
      end
    end
else
  nexa=an;
end
```

```
nex=nexa-mini;
```

A.3.7 Funkcija 8. Vertical and horizontal min-difference

```
%File name: MUNK.M
%Create: V.,LRSS, January 30,1998
%Main programs: LPR.M,MATT.M
function [a1,a11]=munk(a)
clear miniv minin
len=length(a(1,:));
for i=1:len
  miniv(i)=min(a(i,:));
end
for i=1:len
  a1(:,i)=a(:,i)-miniv';
end
for i=1:len
  minin(i)=min(a1(:,i));
end
for i=1:len
  a11(i,:)=a1(i,:)-minin;
end
```

A.3.8 Funkcija 9. Min of matrix with ignored elements 0

```
%File name: MIOUT0.M
%Create: V.,LRSS, January 30,1998
%Main program: LPR.M
function [miv,ata]=miout0(at0)
global xset minix imin
dil=length(at0(1,:));
for i=1:dil
  for j=1:dil
    if (xset(i)==0)&(xset(dil+j)==0)
      ata(i,j)=at0(i,j);
    elseif (xset(i)==0)&(xset(dil+j)==j)
      ata(i,j)=1000;
    elseif (xset(i)==i)&(xset(dil+j)==0)
      ata(i,j)=1000;
    elseif (xset(i)==i)&(xset(dil+j)==j)
```

```
      ata(i,j)=2000;
    end
  end
end
mi=min(ata);
miv=min(mi);
minix(imin+1)=miv;
imin=imin+1;
```

A.3.9 Funkcija 10. Min of cost function

```
%File name: MINCOS.M
%Create: V.,LRSS, January 30,1998
%Main program: LPR.M
function mc=mincos(am,a)
leng=length(am(:,1));
ss=0;
for i=1:leng
  ss=ss+a(am(i,1),am(i,2));
end
'     Minimum of Cost Function of SMTDP/Permutation    '
mc=1/2*ss;
```

A.4 Program 3. COMPUTE INDEPENDENT ZEROS OF MATRIX

```
%File name: MATT.M
%Created: V., LRSS, February 10, 1998
%Functions: TESTA.M, MUNK.M, NEXTA.M
tma=[2 4 2 6;4 3 2 4;2 4 2 4;3 6 3 9]
global xset ste
len=length(tma(1,:));
xset=zeros(size(1:2*len));
ste=0;
[a1 a11]=munk(tma)
[xset stev]=testa(a11)
a111=nexta(a11,stev)
```

A.5 Program 4. BE HELP BY HANAN'S ALGORITHM

```
%File name:HPOST.M
%Create: V., LRSS,FRI,Feb.12,1998
%Input data
%Function: HANA.M
dat1=[1 6;3 8;4 3;4 5;6 2;6 4;7 7];
a1=hana(dat1)
dat2=[1 6;2 6;3 6;3 8;4 3;4 5;6 2;6 4;7 7];
a2=hana(dat2)
dat3=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;6 2;6 4;7 7];
a3=hana(dat3)
dat4=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;5 3;6 2;6 4;7 7];
a4=hana(dat4)
dat5=[1 6;2 6;3 6;3 8;4 3;4 5;4 6;5 3;6 2;6 4;7,4;7 7];
a5=hana(dat5)
```

A.5.1 Funkcija 11. Compute partition blocks

```
%File name:HANA.M
%Create:J.V.,LRSS,FRI,Feb.11,1998
function polb=hana(dati)
dl=length(dati);
dolg=max(max(dati)); %dimension of mini-rectangle
pol=zeros(dolg);
cont=pol(1,:);
for i=1:dolg
  for j=1:dolg
    for k=1:dl
      if dati(k,:)==[i j]
        pol(i,j)=1;
      end
    end
  end
end
pol1=pol %i dimension is y-axe and j is x-axe
for x=1:dolg
  for y=1:dolg
```

```
    if pol1(x,y)==1
      poly(x,y)=y;
    else
      poly(x,y)=0;
    end
  end
end
a=max(poly');
pola=poly
for x=1:dolg
  for y=1:dolg
    if pola(x,y)==0
      pola(x,y)=1000;
    end
  end
end
b=min(pola');
for x=1:dolg
  if b(x)==1000
    b(x)=0;
  end
end
razli=a-b; % number of connections in block of partition P
polb=pol1;
for x=1:dolg
  con=razli(x);
  for y=1:dolg
    if con>=1
      if polb(x,y)==1
        polb(x,y+1)=1;
        con=con-1;
      end
    end
  end
end
pol1; %flag-field of vertices
polb; %blocks of partition into y direction
```

A.6 Program 5. SHORTEST ROUTE BETWEEN TWO POINTS IN FIELD WITH BARS

```
%Processing of Lee's algorithm
%File name: LEEA.M
%Create: V.,LRSS,FRI,Feb.15,1998
%Functions: PLUS1.M,CORBAR.M
%begin of route:111
%end of route:333
%bar:222
%without of manhattan unit:0 (lengths out of routing problem)
%
clear
global mfild fdo dol fpj fpoj
%input and output point
fpoi=[1 4];
fpoj=[5 4];
%bars
ome1=[1 6];
ome2=[3 3;3 4;3 5;3 6];
ome=[ome1; ome2]; %all bars together
%field dimension
fdo=7;
%
fpi=fpoi(1);
fpj=fpoj(1);
mfild=zeros(fdo);
mfil=mfild;
dol=length(ome);
%make 1 abot 222
mfild(fpi,fpoi(2))=111;
for i=fpi:fdo
  for j=1:fdo
    if mfild(i,j)==111
      if j+1<=fdo
        mfild(i,j+1)=1;
      end
      if j-1>=1
        mfild(i,j-1)=1;
```

```
      end
      if i+1<=fdo
        mfild(i+1,j)=1;
      end
    end
  end
end
mfild;
%field filling wit manhattan units
maxd=2*fdo;
for k=2:maxd
  plus1(k);
end
%insert bars
for k=1:dol
  vik=ome(k,:);
  mfild(vik(1),vik(2))=222;
end
%correction under bars
numb=0; %number of bars in field
for i=1:fdo
  for j=1:fdo
    ddi=(mfild(i,j)==222)&(i>fpoi(1)); %bar out of level 111
    for q=1:fdo-j
      if mfild(i,j+q)==222
        rez(i,j)=q;
      else
        break;
      end
    end
    re=max(max(rez)); %length of bar
    frame=(j-1>=1)&(i+1<fdo)&(j+re<=fdo)&(mfild(i,j)==222);
    if frame
      numb=numb+1
      mbar=[mfild(i,j-1:re+j); mfild(i+1,j-1:j+re)];
      poz=[i j];
      if numb==1
        mbar1=mbar
        poz1=poz
```

```
      elseif numb==2
        mbar2=mbar
        poz2=poz
      elseif numb==3
        mbar3=poz
        poz3=poz
      end
    else
    end
  end
end
mbar=mbar2;
poz=poz2;
%correction after bar
%correction just after bar
mab=corbar(mbar);
dlt=size(mab);
for i=1:fdo
  for j=1:fdo
    cori=(i==poz(1)+1)&(j>=poz(2))&(j<poz(2)+re);
      if cori
        mfild(i,j)=mab(2,j-1);
      else
      end
  end
end
%advanced field corrections
fmab=mab(dlt(1),2:dlt(2)-1);
for q=1:fdo-poz(1)-1
  fma(q,:)=fmab+q;
end
fma;
for i=1:fdo
  pog=(i>=poz(1)+2);
  if pog
    mfild(i,poz(2):poz(2)+re-1)=fma(i-poz(1)-1,:);
  end
end
%length of route
```

```
roul=mfild(fpoj(1)-1,fpoj(2));
for i=1:fdo
  for j=1:fdo
  lopo=(mfild(i,j)>roul)&(mfild(i,j)<111);
    if lopo
      mfild(i,j)=0;
    end
  end
end
mfild(fpoj(1),fpoj(2))=333;
%corrections for bar in line with input point
for j=1:fdo
  if j>poz1(2)
    mfild(1,j)=mfild(2,j)+1;
  end
end
%last field matrix
mfild
```

A.6.1 Funkcija 13: Put manhattan-unit a into field

```
%File name:PLUS1.M
%Create: V.,LRSS,FRI,FEB.15,1998
%Main program: LEEA.M
function mfild=plus1(a)
global mfild fdo dol fpj
for i=1:fdo
  for j=1:fdo
    if mfild(i,j)==a-1
      if j+1<=fdo
        if mfild(i,j+1)==0
          mfild(i,j+1)=a;
        end
      end
      if i+1<=fdo
        if mfild(i+1,j)==0
          mfild(i+1,j)=a;
        end
      end
```

```
      if j-1>=1
        if mfild(i,j-1)==0
          mfild(i,j-1)=a;
        end
      end
      if i-1>=1
        if mfild(i-1,j)==0
          mfild(i-1,j)=a;
        end
      end
    else
    end
  end
end
```

A.6.2 Funkcija 14. Correction field-part behind bar

```
%File name:CORBAR.M
%Create: V.,LRSS,FRI,Feb.17,1998
%Main program: LEEA.M
function corm=corbar(matbar)
global fdo
dbar=size(matbar);
for j=1:dbar(2)
  if j==1
    a=matbar(dbar(1),j);
    av(1)=a;
  else
    a=a+1;
    av(j)=a;
  end
end
for j=1:dbar(2)
  if j==1
    b=matbar(dbar(1),dbar(2));
    bv(dbar(2))=b;
  else
    b=b+1;
    bv(dbar(2)-j+1)=b;
```

```
  end
end
abc=[av;bv];
abm=min(abc);
corm=[matbar(1,:);abm];
```

Dodatek B

VPRAŠANJA IN ODGOVORI

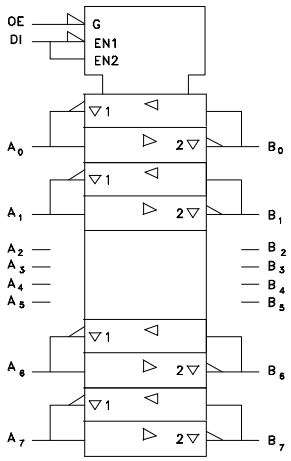
Da bi bralec lahko preveril, ali je od pregleda oziroma branja učbenika, kaj odnesel, v nadaljnjem besedilu postavjamo zanj najprej nekaj vprašanj, nato pa, ločeno, ustrezne odgovore. Odgovor naj bralec pogleda le tedaj, ko je že sam prišel do odgovora na določeno vprašanje in ga zanima, ali je njegov odgovor skladen z našim odgovorom.

B.1 VPRAŠANJA

Vpr. 1: Dan je IEC-simbol logične sheme na sliki B.1. Ugotovite, za kakšno funkcionalnost logične sheme gre? Razdelajte dano shemo detajlno (če je mogoče, na nivoju logičnih vrat), vendar še vedno z ustreznimi IEC-simboli. V katalogu za MSI MOS-gradnike poiščite MSI-čip, ki v celoti pokriva funkcionalnost simbola na sliki B.1. (vprašanje sodi v okvir poglavja 2)

Vpr. 2: Napišite matriko permutacij reda 5×5 tako, da ima ničle tako razporejene, da sploh ni potrebno napraviti računa optimizacije neodvisnih ničel. (vprašanje sodi v okvir poglavja 2)

Vpr. 3: Ena od logičnih shem na slikah v tretjem poglavju ustreza incidenčni matriki I, če so stolpci izhodi in vhodi **s** in vrstice imena logičnih operatorjev **a**.



Slika B.1: IEC-simbol logične sheme.

$$\mathbf{s} = (f, x, y_1, y_2, y_3, y_4, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8)$$

$$\mathbf{a} = (a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8)$$

$$I = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}.$$

Najdite sliko z ustrezno shemo. (vprašanje sodi v okvir poglavja 3)

Vpr. 4: Zakaj lahko množico vseh modulov v modularni shemi lahko predstavljamo s Booleovo konstanto 1? (vprašanje sodi v okvir poglavja 3)

Vpr. 5: Ali se da način delovanja žice na vhodu logičnega operatorja ali med takšnimi operatorji predstaviti z načini delovanja negatorja? (vprašanje sodi v okvir poglavja 3)

Vpr. 6: Kakšna je razlika med pisanimi in tiskanimi velikimi črkami na programirnem listu na sliki 4.17? (vprašanje sodi v okvir poglavja 4)

Vpr. 7: Kakšna je bistvena razlika med PAL- in GAL-gradniki? (vprašanje sodi v okvir poglavja 5)

Vpr. 8: Izračunajte izhodno funkcijo X na sliki 6.6, če sta selekcijska vhoda S0 in S1. Če imate na voljo LCA MACRO LIBRARY, XACT poiščite ustrezni izhodni MX in preverite, če je vaš rezultat pravilen. (vprašanje sodi v okvir poglavja 6)

Vpr. 9: Kakšna je razlika med zakasnitvijo in zamuditvijo v logični shemi? (vprašanje sodi v okvir poglavja 7)

Vpr. 10: Kakšno zakasnitev ima povezava na notranji strani 4-stranske kartice velikosti 15×15 cm, če se priključka povezave nahajata v diagonalnih kotih? (vprašanje sodi v okvir poglavja 9)

Vpr. 11: Napišite v Iversonovi notaciji, v skladu z izrazom (10.1), izhodni vektor $\mathbf{y}$ v popolni Shefferjevi normalni obliki. (vprašanje sodi v okvir poglavja 10)

Vpr. 12: Ali je preslikava na sliki 10.5 Mealyjeve ali Mooreove narave? (vprašanje sodi v okvir poglavja 10)

Vpr. 13: Med matriko (10.11) in sistemom izrazov (10.12) je ta razlika, da v omenjenem sistemu izrazov ne najdemo spremenljivke e. Ali bi znali to spremenljivko dodati v (10.12), da vsi izrazi ohranijo svojo opredelitev? Kaj pravzaprav pomeni to vnašanje spremenljivke e v omenjene izraze? (vprašanja sodijo v okvir poglavja 10)

Vpr. 14: Kakšna je kompleksnost C za logično shemo na sliki 4.8, če upoštevamo programirne elemente F in drugič, če teh ne upoštevamo? (vprašanje sodi v okvir poglavij 4 in 10)

Vpr. 15: Ali znate opisati notacijo programirnega gradnika XC4010-5PG191C. Podajte informacije, ki jih podaja ta notacija. Pri tem si lahko pomagate s ustreznim priročnikom The Programmable Logic Data Book, 1993. (vprašanje sodi v okvir poglavja 10)

B.2 ODGOVORI

Odg. 1: Detajlno izrisani IEC-simbol vidimo na sliki B.2. Vidimo, da gre za 8-kanalni 3-stanjski vhodno-izhodni prenosnik vodila. MSI-čip s takšno vsebino je na primer PC74C/HCT640. Glede na ta čip velja: `OE=`$\overline{\texttt{OE}}$, `DI=DIR`, `Ai=A`$_\texttt{i}$ in `Ai=B`$_\texttt{i}$.

Odg. 2: Ničle so porazdeljene v diagonali matrike.

$$A = \begin{bmatrix} 0 & x & x & x & x \\ x & 0 & x & x & x \\ x & x & 0 & x & x \\ x & x & x & 0 & x \\ x & x & x & x & 0 \end{bmatrix}$$

x je lahko kakršnokoli pozitivno celo število ali tudi 0.

Odg. 3: Incidenčni matriki ustreza slika 3.6.

Odg. 4: Zato, ker, če iz takšne množice izhajamo pri neki vhodni črki, dobimo v vseh primerih enak rezultat.

Odg. 5: Se da, kot na primer vidimo na naslednji sliki B.3.

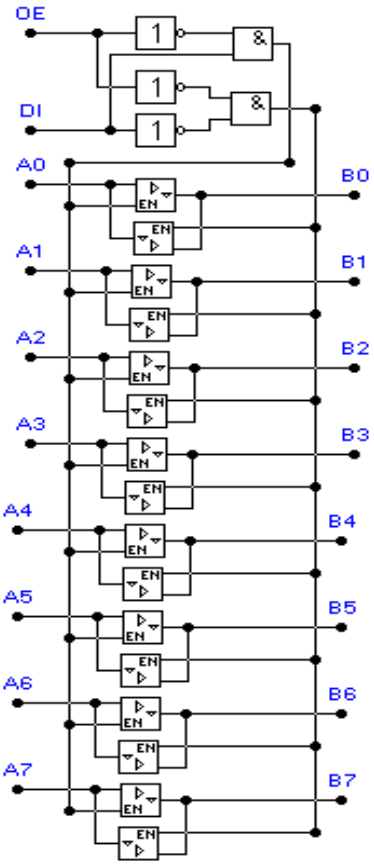
Odg. 6: Za tiskane črke je odgovoren proizvajalec programirnega gradnika, medtem ko so pisane črke programski izraz načrtovalca.

Odg. 7: PAL-tehnika uporablja vrsto različnih čipov, medtem ko GAL-tehnika ponuja v pricipu en čip, v okviru katerega s programiranjem dosežemo značilnosti kateregakoli PAL-čipa (GAL-čip imamo lahko za virtualni PAL-čip). Seveda poleg tega GAL-tehnika ponuja še vrsto drugih posebnosti, kar ni v relaciji s PAL-tehniko.

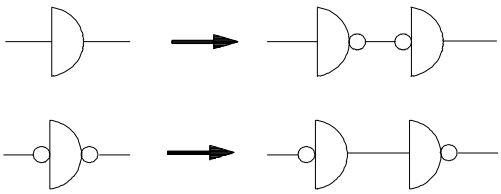
Odg. 8: Izhodna funkcija X je

$$X = \bar{S}_1(S_0 G \vee \bar{S}_0 F) + S_1 Q.$$

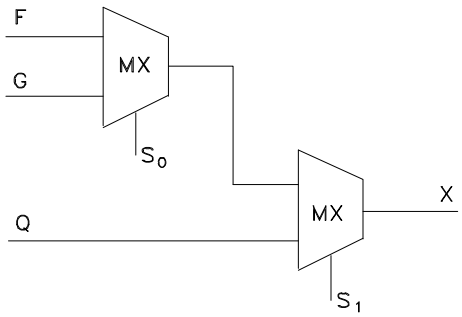
Izraz izhaja iz logične sheme na sliki B.4.



Slika B.2: Detajlnejša razdelava IEC-simbola na sliki B.1.



Slika B.3: Odgovor na vprašanje 4.



Slika B.4: Multipleksorska shema za funkcijo X.

Odg. 9: Odgovor izhaja iz slike 7.1. Spodnji primer na tej sliki prikazuje, da pride zaradi inercije vezja do zamuditve vhodne špice (dinamičnega signala).

Odg. 10: Iz spiska raznih zakasnitev (prva stran poglavja 9), izhaja, da je poprečna zakasnitev 4-stranske kartice na notranji strani 180 ps/palec. Tako je za omenjeno povezavo izračunana poprečna zakasnitev 1.5 ns.

Odg. 11: Popolna Sheferjeva oblika v vektorski izražavi je

$$\mathbf{y} = (\bar{\mathbf{x}}| \equiv \overline{D'})||K.$$

Odg. 12: Preslikava je Mooreove narave, ker je stanje avtomata dodeljeno izvajalnemu operatorju v diagramu poteka.

Odg. 13: Spremenljivko e disjunktivno dodamo v vsak izraz za naslednje stanje. Zadnje pomeni, da je lahko tudi $\mathrm{D}^1 a_i = a_i$.

Odg. 14: Kompleksnost sheme je v obeh primerih enaka $C = 1$. Kompleknost je maksimalna, ker velja

$$C_1 = \sum_1^1 \frac{10}{1 \cdot 10} = 1$$

$$C_2 = \sum_1^1 \frac{2}{1 \cdot 2} = 1$$

Pravzaprav je $C = 1$ tudi zato, ker imamo samo eno izhodno funkcijo.

Odg. 15: Notacija nam pripoveduje, da gre za Xilinx gradnik z 10000 ekvivalentnih vrat, hitrostne stopnje -5 (speed grade) z zakasnitvami v CLB 2 do 12 ns, s PG-čipno postavitvijo s 191 priključki ter na koncu C, ki pomeni temperaturno področje uporabe med 0 in 70 ^{o}C.

Literatura

[1] J. Virant, Logične osnove odločanja in pomnjenja v računalniških sistemih. ZAFER 1996, 381 str.

[2] J. Virant, L. Pipan, Grafični simboliza logične sheme. Iskra, Standard, Ljubljana 1986, str. 174 (prevod).

[3] MATLAB, Optimization Toolbox, User's Guide, Version 5, MathWorks, Inc., 1996.

[4] J. Munkres, Algorithms for Assignment and Transportation Problems, J. SIAM, vol. 5, marec 1957, str. 32-38.

[5] L. Steinberg, The Backboard Wiring Problem: Placement Algorithm, SIAM Review, vol. 3, 1961, str. 37-50.

[6] M. Hanan, New Wiring for Large Scale Integrated Circuits, IBM Research Report RC - 1975, februar 1965.

[7] M. A. Breuer, Design automation of digital systems, Theory and techniques. Prentice-Hall, Inc., EnglewoodCliffs, NJ, 1972.

[8] System Overview, CADAM Inc., 1991.

[9] J. Virant, Logično snovanje računalniških struktur in sistemov, FER, Ljubljana, 1987.

[10] J. D. Ulman, P. Weiner, Modular Networks and Nondeterministic Sequential Machines. Trans. on EC, IEEE, vol. 21. October 1972.

[11] M. M. Newborn, T. F. Arnold, Universal Modules for Bounded Signal Fan-Out Synchronous Sequential Circuits. Trans. on EC, IEEE, vol 21, January 1972.

[12] V. Guštin, ABEL in programabilna logična vezja, priročnik za vaje. Založba FE in FRI, Ljubljana 1996.

[13] ALTERA, EPLD Handbook, Altera Corporation, 1985.

[14] XILINX, Handbook, Xilinx, Inc.,1986.

[15] M. J. S. Smith, Application-Specific Integrated Circuits. Addison-Wesley, Reading, Harlow, MenloPark, Berkeley, Don Mills, Sydney, Bonn, Amsterdam, Tokio, Mexico City, 1997.

[16] D. Hübner, E. Schönherr, Diagnostik in der Digitaltechinik. VEB Verlag, Berlin 1982.

[17] H. W. Johnson, M. Graham, High-Speed Digital Design, A Handbook of Black Magic. Prentice Hall, Inc., Upper Saddle River, NJ, 1993.

[18] W. Giloi, H. Liebig, Logische Entwurf digitaler Systeme. Spinger-Verlag Berlin-Deidelberg-New York 1973.

[19] J. Virant, Zanesljivost računalniških sistemov. ZAFER-Ljubljana, Ljubljana 1993.

[20] H. A. Sholl, Shou-Chung Yang, Design of Asynhronous Sequential Network Using Read-Only Memories. Trans. on EC, No. 2, vol C-24, February 1975.

[21] F. Belina, D. Hogrefe, A. Sarma, SDL with Application from Protocol Specification. Hanser Verlag, Prentice Hall Int. Ltd., Dotesios Lmt, 1991.

[22] E. Mandado, J. Marcos, S. A. Pérez, Programmable Logic Devices and Logic Controllers. Prentice Hall, London, NY, Toronto, Sydney, Tokyo, Singapore, Madrid, Mexico City, Munich, 1996.

[23] K. Kronlöf, Method Integration, Concept and Case Studies. John Wiley & Sons, Chichester, N.Y., Brisbane, Toronto, Singapore, 1993.

[24] G. De Micheli, R. K. Gupta, Hardware/Software Co-Design. IEEE Proc., Vol. 85, No. 3, marec 1997, str. 349-365.

[25] XILINX, The programmable Logic Data Book, Xilinx,1994.

Stvarno kazalo

[0] Učbenik je urejen z orodjem SWP30. Hrošče in skrivalnice LaTeXa v tem orodju je lovil in odkrival as. Iztok Lapanja dipl. ing.