Univerza v Ljubljani
Fakulteta za elektrotehniko

Gregor Papa

IZVEDBA VGRAJENEGA SAMOTESTA V OKVIRU VISOKONIVOJSKE SINTEZE

Diplomska naloga
(dopolnjena različica)

Mentor: prof. dr. Baldomir Zajc, dipl. ing.

Ljubljana, februar 1997

Povzetek

V tem delu je opisan postopek sinteze integriranih vezij z vgrajenim samotestom. Značilnost takih vezij je, da se med samim delovanjem sproti preverjajo ter tako sama ugotovijo, kdaj je v njih prišlo do napake. V ta namen imajo vezja poleg algoritma za izračun izhodnih podatkov na osnovi vhodnih vgrajen še dodaten testni algoritem, ki preračuna vnaprej postavljene vhodne vektorje in nato preko pričakovanih rezultatov pove, ali so podatki preračunani prav ali ne. To lahko sklepamo, ker v primerih, ko pride do napake v testnem izračunu, zelo verjetno nastane napaka tudi v dejanskem izračunu, saj uporabljata oba algoritma iste funkcijske enote in ostale dele vezja.

V uvodnem poglavju so predstavljena posamezna programska orodja oziroma programski paketi, ki so potrebni za zgoraj omenjeno sintezo. Opisan je programski jezik VHDL, programski paket Amical za visokonivojsko sintezo ter simulacija napak za ovrednotenje rezultatov sinteze.

V drugem poglavju je opisanih nekaj osnovnih značilnosti visokonivojske sinteze. To je načrtovalni proces, katerega izdelek je načrt na RT nivoju. Struktura je sestavljena iz dveh delov; podatkovne in krmilne poti. Podatkovna pot je povezava funkcijskih enot, ki izvajajo aritmetične in logične operacije, registrov za hranjenje podatkov ter vodil za prenos podatkov. Krmilna pot pa je logika, ki krmili prenos podatkov po podatkovni poti. Glavne naloge visokonivojske sinteze

so: prevajanje, transformacija grafa toka podatkov, razvrščanje, dodeljevanje in povezovanje.

Tretje poglavje govori o pomenu in načinu izvedbe hkratnega testiranja v integriranih vezjih. Predstavljene pa so tudi prednosti in slabosti takega načrtovanja.

Celotno četrto poglavje je namenjeno praktičnemu prikazu postopka za sintezo samotestabilnega integriranega vezja. Predstavljen je primer diskretnega proporcionalno-integrirno-diferencirnega regulatorja, ki uravnava delovanje različnih procesov. Opis poteka načrtovanja je razdeljen v posamezne korake. Leti vsebujejo razlago in slike. Končno vezje je nato ovrednoteno še s simulacijo napak, ki prikaže, kako velika pridobitev za zanesljivost vezja je vgrajeni samotest.

V končnih poglavjih so opisane še pridobitve, ki so nastale s takšno izvedbo integriranega vezja, ter možne razširitve in izboljšave. Učinkovitost je mogoče povečati predvsem z boljšimi testnimi vektorji ter s spremenjeno razporeditvijo uporabe posameznih registrov.

Kazalo

1 Programsko okolje

Pri delu sem uporabil več različnih programskih orodij ter algoritmov za reševanje posameznih problemov. Vezje sem opisal v jeziku VHDL, ki je precej uveljavljen in za takšne opise tudi izdelan programski jezik. Samo obdelavo vezja sem izvršil s programskim paketom Amical, dobljeni rezultat pa ovrednotil s simuliranjem napak.

Da bodo uporabljeni pojmi in izrazi razumljivejši, naj na kratko predstavim programski jezik ter uporabljeni programski paket.

1.1 Programski jezik VHDL

VHDL (Very high speed integrated circuits Hardware Description Language) je standardni industrijski programski jezik za opis digitalnih logičnih vezij. Standardiziran je od leta 1987 in je prenosljiv na različna programska orodja. Namenjen je predvsem analizi in modeliranju digitalnih elektronskih vezij in sistemov.

Skladno z rastjo kompleksnosti današnjih vezij se veča potreba po uporabi takega opisnega jezika. Jezik za opis vezij omogoča opisovanje na višjem, abstraktnejšem nivoju. Načrtovanje s splošnim jezikom pa omogoča preprosto pretvarjanje vezij iz ene tehnologije v drugo ter hkrati preverjanje in ocenjevanje posameznih različic [1].

Opise, narejene v VHDL, se da simulirati na komercialno dostopnih simulatorjih in se jih lahko uporabi za opis še tako kompleksnega sistema. VHDL namreč podpira več stopenj posploševanja in opisovanja: funkcijski opis (behavioral), strukturni opis (structural) ter opis toka podatkov (data-flow).

Pri funkcijskem opisu se stavki izvajajo sekvenčno (procesi, funkcije in procedure), pri strukturnem in podatkovnem opisu pa se vsi dogodki odvijajo sočasno ne glede na vrstni red opisa.

V najpreprostejši obliki je opis komponent v VHDL sestavljen iz opisa vmesnika (interface) in opisa zgradbe (architecture). Opis vmesnika se prične z ukazom ENTITY in vsebuje vhodno-izhodne priključke. V tem delu so lahko opisane tudi druge zunanje lastnosti komponent, kot so časovne in temperaturne odvisnosti. Opis zgradbe se prične z ukazom ARCHITECTURE, ki določa funkcionalnost komponente. Ta funkcionalnost je odvisna od vhodno-izhodnih signalov in ostalih parametrov, ki so definirani v opisu vmesnika. Funkcijski opis komponente se prične z ukazom BEGIN.

Primer opisa v VHDL:

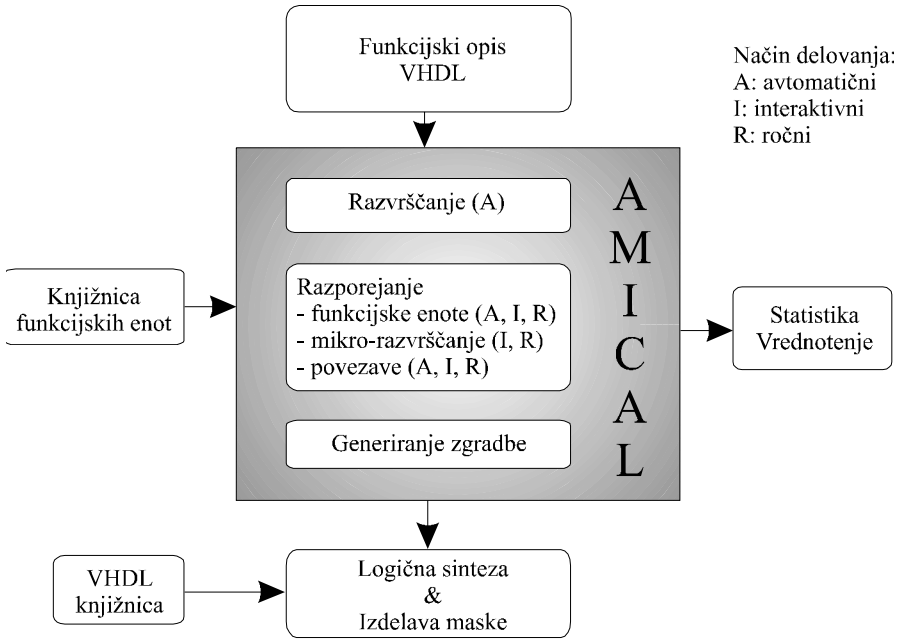
```
ENTITY ime_komponente IS
   vhodni in izhodni priključki.
   fizični in ostali parametri.
END ime_komponente;

ARCHITECTURE vrsta_opisa OF ime_komponente IS
   deklaracije
BEGIN
   opis funkcionalnosti komponente s stališča njenih vhodov
   ter vplivov zaradi fizičnih in ostalih parametrov.
END vrsta_opisa;
```

Za simulacije, v katerih se preverja funkcionalnost sistema, se uporablja funkcionalni opis. V simulacijah, kjer se preverjajo časovni odzivi, pa se uporablja opis na strukturnem nivoju.

1.2 Načrtovalni paket Amical

Amical je komercialen programski paket, izdelan na francoskem institutu TIMA v Grenoblu. Je programsko orodje za načrtovanje vezij do RT (register transfer) nivoja. Kot vhod uporabi funkcijski opis v VHDL, izvede razvrščanje (scheduling) in dodeljevanje (allocation) ter izdela izhodno zgradbo, sestavljeno iz podatkovnih poti (data-path) in krmilnika (controller), kar lahko uporabimo v povezavi z orodji za sintezo, ki delujejo na logičnem ter RT nivoju.

Amical uporabi kot vhod dve vrsti informacij: funkcijski opis v VHDL ter zunanjo knjižnico funkcijskih enot (slika 1.1).



Slika 1.1: Potek načrtovanja v Amicalu

Podrobnejši potek celotnega načrtovanja v Amicalu prikazuje slika v prilogi 3.

Uporabljena različica programa zajema skoraj celotno množico VHDL sekvenčnih stavkov (wait, loop, exit...) za funkcijski opis. Knjižnica lahko vsebuje standardne funkcijske enote (seštevanje, odštevanje...) ali pa kompleksne funkcijske enote, kot so pomnilnik, predpomnilnik, vhodno/izhodne enote, itd.

Povezave med funkcijskimi enotami ter operacijami so izvedene z imeni. Operacije teh enot so uporabljene preko VHDL funkcij in procedur.

Knjižnica enot vsebuje opis vsake funkcijske enote, ime enote in nekatere fizične parametre ter vhodno/izhodne priključke. Vsaka operacija, ki jo funkcijska enota izvede, pa je še dodatno razbita v posamezne osnovne cikle (urine cikle oziroma krmilne korake).

Amical je organiziran kot interaktivno okolje, kjer se lahko prepletata avtomatična in ročna sinteza. V interaktivnem načinu delovanja lahko po korakih spremljamo delo, ki ga sicer program opravi v avtomatičnem načinu načrtovanja. Odzivni čas je zelo kratek, kar daje programu še posebno odliko. Kombinacija avtomatičnega in ročnega načrtovanja namreč omogoča hitro in obsežno raziskavo načrtovalnega prostora v realnem času. Poleg tega pa Amical nudi vrsto dodatnih pripomočkov za analizo zgrajene strukture (statistika, vrednotenje, povezave med VHDL opisom ter zgradbo...).

1.3 Simulacija napak

Medtem ko se logična simulacija uporablja pri načrtovanju vezja, se simulacija napak uporabi za izdelavo in preverjanje testnih vzorcev, ko je vezje že zgrajeno. Simulacija napak ima pomembno vlogo pri računalniško podprtem načrtovanju digitalnih sistemov. To še posebej velja za načrtovanje zelo obsežnih ter zelo zanesljivih sistemov. Simulacija napak je simuliranje sistema ob prisotnosti namerno vnesenih napak ter ugotavljanje učinkovitosti testnih naborov za zaznavanje teh napak [2].

Zaradi tega je namen simulacije:

- preverjanje množice testnih vektorjev oziroma ugotavljanje pokritosti množice napak z določeno testno množico,

- preverjanje delovanja vezij oziroma sistemov pri pogojih, ki pri načrtovanju niso bili predvideni,

- izdelava zbirk za določanje mesta napak. Te zbirke vektorjev in izhodov se primerja z izhodi sistema, ko ga testiramo in tako določimo napako.

Obstaja več nivojev simuliranja:

- arhitekturna simulacija: je najvišji nivo (opis ima najmanj podrobnosti) simulacije napak in se uporablja za simuliranje delovanja arhitektur sistema. Pri tem niso simulirane logične vrednosti signalov, temveč pretok podatkov v sistemu (določanje ozkih grl pri pretoku podatkov);

- funkcionalna simulacija: je najvišji nivo logične simulacije. Vezje je predstavljeno s funkcionalnimi bloki, delovanje posameznih funkcionalnih blokov pa je opisano v računalniškem opisnem jeziku (VHDL);

- simuliranje na nivoju vrat: tu lahko določamo vrednosti logičnih signalov, pa tudi časovne karakteristike. Osnovni gradniki so logična vrata;

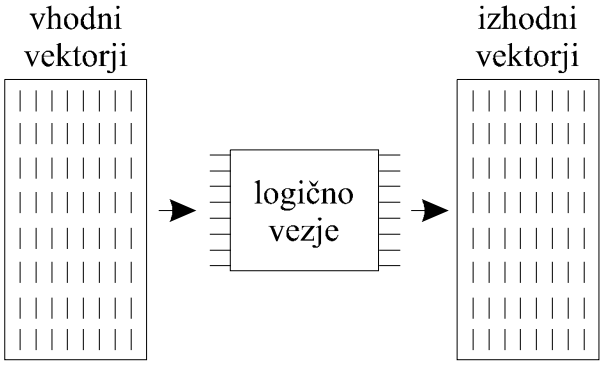
- tranzistorski nivo: je najnižji nivo simulacije. Simuliramo posamezne fizične elemente v vezju. Modelirane napake na tem nivoju so fizikalne, vendar jih lahko modeliramo tudi na višjih nivojih. Ta nivo simuliranja je pri velikih sistemih preobsežen.

Pri simulaciji napak v glavnem modeliramo dve vrsti napak. To sta zadrgnitev na 0 (stuck-at-0) ter zadrgnitev na 1 (stuck-at-1). V tem primeru gre za napako (lahko nastalo že med proizvodnjo ali pa kasneje med delovanjem), kjer je izhod oziroma vhod nekega elementa v vezju na stalnem logičnem nivoju 0 ali 1. To pa lahko bistveno spremeni delovanje vezja.

Preden je integrirano vezje ali plošča dokončana, se torej preveri, če vsebuje kakršnekoli napake. To se izvede tako, da se na vhod testiranega elementa v določenih časovnih trenutkih postavi vnaprej določene vektorje, prav tako pa se v določenih trenutkih bere stanje izhodnih vektorjev, ki se jih nato primerja s pričakovanimi izhodnimi vektorji glede na vhodne (slika 1.2).

Pri tem je zelo pomembno narediti tak niz vhodnih vektorjev, ki bo odkril čim več ali pa kar vse možne modelirane napake na elementu. Ta niz pa naj bo čim

manjši, da bo kratek tudi čas, potreben za ugotovitev morebitnih napak; hkrati pa ne sme biti premajhen, da se ne bi kakšna napaka izognila testu ter tako odšla skupaj z izdelkom v prodajo.



Slika 1.2: Princip delovanja simulatorja napak

Namen simulacije napak je preverjanje zmožnosti zaznavanja simuliranih napak z danimi testnimi vektorji. Tako se izvede več logičnih simulacij na vezju z vnesenimi napakami. Simulator primerja izhode vezja z napakami ter vezja brez napak (njegove vrednosti so referenčne). Če ima vezje z napakami enake izhode kakor referenčno vezje, potem velja napaka za neodkrito. Če pa se nek vektor ne sklada z referenčnim, velja napaka za odkrito.

Kakovost testnih vzorcev je ponavadi opisana s pokritostjo napak ali odkrivnim razmerjem (fault coverage). To je razmerje števila odkritih simuliranih napak s številom vseh simuliranih napak. Tako 100% pokritost pomeni, da je testni vzorec sposoben odkriti vse napake. V splošnem stopnja pokritosti 95% ali več velja za zelo dobro.

2 Visokonivojska sinteza

2.1 Principi visokonivojske sinteze (High-level synthesis)

To je načrtovalni proces, katerega produkt je načrt na RT nivoju, ki vsebuje pretvorjen funkcijski opis digitalnega sistema. Struktura je sestavljena iz dveh delov: podatkovne in krmilne poti. Podatkovna pot je povezava funkcijskih enot, ki izvajajo aritmetične in logične operacije, registrov za hranjenje podatkov ter vodil za prenos podatkov. Krmilna pot pa je logika, ki krmili prenos podatkov po podatkovni poti [3].

VLSI (Very Large Scale Integration) tehnologija je že pred leti dosegla zelo gosta integrirana vezja. Sisteme s tako kompleksnostjo je težko načrtovati ročno, pa naj gre za postavljanje tranzistorjev ali pa za definiranje signalov. S kompleksnostjo sistemov raste tudi potreba po avtomatizaciji načrtovanja na abstraktnejših nivojih, kjer sta funkcionalnost ter iskanje kompromisnih rešitev razumljivejši [4].

Začetki visokonivojske sinteze segajo v 60-ta leta. V 70-tih letih je bila pozornost usmerjena v avtomatizacijo sinteze na nižjih nivojih abstrakcije. Tedaj je bil narejen tudi velik napredek v razvoju algoritmov in tehnik. V 80-tih letih so se dosežki raziskav na področju visokonivojske sinteze pričeli prenašati v

industrijska okolja. Danes pa že uporabljamo visokonivojsko sintezo pri načrtovanju integriranih vezij.

Industrija išče tiste produkcijske načine, ki bodo povečali produktivnost ter zvečali konkurenčnost. Tako se je v devetdesetih letih avtomatizacija celotnega načrtovalnega procesa razširila na vse faze načrtovanja, od zamisli pa do končne izvedbe.

Razvoj avtomatizacije je pomemben zato, ker zahteva industrija zelo kratek čas, potreben od zamisli do tržišča. Ker je vsaka ponovna predelava izdelka ter njegovo izboljševanje zelo drago, se pojavlja težnja po vzpostavitvi končnega izdelka že v prvi iteraciji načrtovanja. Za to pa so potrebna CAD (Computer Aided Design) orodja za preizkušanje funkcionalnosti in načrtovalnih pravil celotnega integriranega vezja. Za dosego pravilnosti že ob prvem načrtovanju pa je potrebno natančno modeliranje procesa ter natančno ocenjevanje kvalitete. Te ocene so seveda potrebne že v zelo zgodnji fazi načrtovanja.

Avtomatizacija dela ali pa celotnega procesa ima več prednosti, ki postanejo izrazitejše pri pomiku avtomatizacije na višje nivoje. Avtomatizacija zagotavlja mnogo krajši načrtovalni cikel. Omogoča raziskavo različnih tehnologij načrtovanja, saj se posamezne rešitve zelo hitro preračunajo. Končno pa, če je algoritem sinteze dobro razumljiv, lahko avtomatična orodja prekosijo človeka pri izdelavi zelo kvalitetnih načrtovanj. Vendar pa preverjanje pravilnosti teh algoritmov ter CAD orodij ni preprosto. CAD orodja se še vedno ne morejo primerjati s človeško kvaliteto pri avtomatizaciji celotnega procesa, čeprav je pri posameznih nalogah ta kvaliteta lahko dosežena. Glavni oviri pri tem sta: problemi velikih razsežnosti, kjer je potrebno zelo zmogljivo preiskovanje prostora, in podrobni načrtovalni modeli, ki potrebujejo posebne algoritme, sposobne zadostiti mnogim ciljem ter zahtevam [4].

Sintezo načrtovanja lahko označimo kot prevedbo procesa iz funkcijskega opisa v strukturni opis, podobno, kakor je prevajanje visokega programskega jezika, kot je C ali Pascal, v zbirnik. Te posamezne vrste predstavitev oziroma opisov pa imajo tudi svoje značilne ter lastnosti.

Nivo	funkcijska predstavitev	strukturna predstavitev	fizična predstavitev
sistemski nivo	tok podatkov algoritmi	procesorji krmilniki pomnilniki vodila	plošče integrirana vezja
mikroarhi-tekturni nivo	registrski prenosi	ALE množilniki registri	integrirana vezja
logični nivo	Boolove enačbe	vrata flip-flopi	celice
nivo vezja	prenosne funkcije	tranzistorji povezave	tranzistorska postavitev stiki

Tabela 2.1: Značilnosti posameznih predstavitvenih nivojev

Pri funkcijskem opisu je pomembno, kaj model dela, in ne, kako je sestavljen. Model obravnavamo kot črno škatlo z navedenimi vhodi in izhodi ter prenosnimi funkcijami, ki določajo stanja izhodov glede na posamezne vhode in čas. Ta opis vsebuje tudi opis vmesnika ter opis posameznih zahtev, ki se nanašajo na model.

Strukturni opis povezuje funkcijski opis ter fizično predstavitev. Je preslikava funkcijskih opisov v niz komponent in povezav, tako da zadoščajo zahtevam, kot so cena, površina in zastoji.

Fizični opis se ne ozira na to, kaj naj bi model delal in poveže njegovo strukturo v prostoru oziroma na siliciju.

2.2 Sinteza pri posameznih nivojih procesa

Naloga sinteze je, da določi strukturo ciljnega sistema, če sta podana opis njegovega delovanja ter seznam zahtevanih omejitev in ciljev. Načrtovanje je lahko opisano na različnih stopnjah podrobnosti. Podobno je tudi sintezo moč izvajati na različnih nivojih abstraktnosti. Na najnižjem nivoju sinteze se osredotočimo na notranjo zgradbo in delovanje polprevodniških elementov (npr. tranzistorjev). Nivo više je sinteza vezij, kjer se sistem obravnava kot skupek polprevodniških elementov. Temu sledi logični nivo sinteze, kjer je sistem

predstavljen z množico logičnih elementov (npr. vrata), njegovo delovanje pa opisano z logičnimi enačbami. Kadar gledamo na sistem kot na združbo registrov, aritmetičnih logičnih enot (ALE), multiplekserjev (MUX), vodil, govorimo o RT (angl. register-transfer) nivoju sinteze. Na RT nivoju je delovanje sistema podano s prenašanjem in preoblikovanjem podatkov. Naslednji, abstraktnejši nivo, je algoritmični nivo, ki je osredotočen na procesorje ter njihovo preoblikovanje vhodnih podatkov v izhodne. Zgornji nivo sinteze predstavlja sistemski nivo, kjer je sistem podan kot omrežje procesorjev, pomnilnikov ter preklopnih elementov.

Sinteza na sistemskem nivoju se imenuje tudi arhitekturna sinteza. Problematika na sistemskem nivoju zadeva števila in konfiguracijo posamičnih komponent (procesorji, pomnilniki, preklopni elementi), ne pa njihove notranje zgradbe. Arhitekturna sinteza se torej ukvarja s problemi, kot so: določanje kompleksnosti sistema kot funkcije krmilnih in podatkovnih poti, razporejanje nadzora, iskanje primernega nabora asinhronih (pod)procesov, razporejanje pomnilniškega prostora, združevanje registrov, izbiranje primerne stopnje paralelizma s pomočjo širine podatkovnih poti [5].

2.3 Merila kvalitete

Za dobro sintezo je potrebno imeti tudi dobra merila, ki neko dobljeno vezje ovrednotijo. Poleg ugotovitve končnega števila seštevalnikov, registrov in ostalih značilnosti pa je še pomembnejša ocena posameznih kriterijev med samim procesom sinteze.

Ocena mora ugoditi trem kriterijem: natančnosti, točnosti in enostavnosti. Da pa so uporabne, morajo ocene zavzemati tudi napoved uporabljene silicijeve površine, napoved zastojev na povezavah ter učinke tranzistorske in povezavne razvrstitve. Poleg tega morajo ocene napovedati izdelovalne, testirne in vzdrževalne stroške ter odražati vpliv sprememb v procesu izdelave na ceno izdelka in njegov učinek [4].

2.4 Algoritmi visokonivojske sinteze

Splošni algoritmi, uporabljeni pri visokonivojski sintezi, so razporejanje (partitioning), razvrščanje (scheduling) in dodeljevanje (allocating). Razporejanje razdeli funkcijski opis v podopise, da tako zmanjša velikost problema oziroma ugodi nekaterim zunanjim zahtevam, kot so velikost integriranega vezja, število nožic, poraba energije ali maksimalna dolžina povezav. Razvrščanje in dodeljevanje sta del razporejanja. Algoritmi razvrščanja razvrstijo spremenljivke in operacije v časovne intervale, algoritmi dodeljevanja pa jih dodelijo spominskim in funkcijskim enotam.

Razvrščanje in dodeljevanje sta medsebojno ortogonalni dejavnosti, čeprav sta medsebojno prepleteni in tesno povezani, saj morata biti za optimalno zasnovo ciljnega sistema obe primerno rešeni. Razvrščanje priredi vsaki operaciji ustrezne krmilne korake, v katerih se operacija izvrši. Dodeljevanje pa poskrbi za dodelitev materialnih virov posameznim operacijam in podatkom (funkcijske enote, pomnilnik, komunikacijske poti). Značilni cenovni funkciji, ki ju poskušamo ob tem minimizirati, sta čas (število krmilnih korakov) in prostor (materialni viri). Med podnaloge dodeljevanja sodi dodelitev registrov posamičnim podatkom, ki se pojavijo v enem koraku, uporabljeni pa so v drugem, ter morajo biti začasno shranjeni v registru. Tu se pojavi zanimiv problem minimizacije števila registrov, ki si jih podatki delijo. Naslednja podnaloga dodeljevanja je dodelitev funkcijskih enot (ALE, seštevalnik itd.) posameznim operacijam. Seveda si lahko različne operacije delijo funkcijsko enoto, vendar le v primeru, ko se njihova izvrševanja časovno ne prekrivajo. To vodi do problema minimizacije funkcijskih enot. Opazimo, da se na tem mestu pojavi tesna zveza med razvrščanjem in dodeljevanjem. Tretja podnaloga dodeljevanja, dodelitev komunikacijskih poti, se pojavi tedaj, ko je potrebno prenesti podatek z enega mesta na drugo. Za to se uporabljajo vodila in multiplekserji. Vodila so enostavnejša, vendar počasnejša od multiplekserjev. Ponavadi uporabimo kombinacijo obeh. Ko sta razvrščanje in dodeljevanje končani, se prične tretji korak visokonivojske sinteze, tj. sinteza krmilnika, ki poskrbi za delovanje, ki je v skladu z rezultati razvrščanja in

dodeljevanja. Zadnji korak sinteze pa poskrbi za dejansko realizacijo sistema, pri čemer se uporabljajo orodja za logično sintezo ter izdelavo mask (layout) [5].

Razvrščanje torej razporeja elemente vezja glede na čas, medtem ko dodeljevanje upošteva število in razporeditev gradnikov vezja. Če je izvedeno razvrščanje pred dodeljevanjem, doda razvrščenim elementom nekatere nove zahteve, ki morajo biti nato izpolnjene pri dodeljevanju. Podobno se pojavijo dodatne zahteve za razvrščanje, če je prej izvedeno dodeljevanje. Seveda pa razvrščanje in dodeljevanje nista edini operaciji pri visokonivojski sintezi. Potrebni so še mnogi algoritmi za izvajanje in optimizacijo krmilnih, spominskih in funkcijskih enot ter vodil.

Pri različnih vrstah načrtovanja (logično, RT in sistemsko) pa se pojavljajo različni osnovni arhitekturni modeli, s tem pa tudi različni pristopi k načrtovanju končnega opisa [4].

Kombinacijska logika: vse funkcijske enote so bloki kombinacijske logike, katere izhode lahko opišemo kot Boolove izraze vhodov. Funkcijske enote z majhnim številom vhodom pa lahko opišemo tudi s tabelo, ki določa izhode glede na vhodne vrednosti.

Funkcijske enote so lahko izvedene na več načinov (ROM, PLA, multipleksersko vezje, Boolova vrata). Pri tem ima vsaka izvedba nekaj prednosti in slabosti. ROM in multipleksersko vezje se zelo povečata ob dodatku dodatnega vhoda. Veliko število vhodov pri ROM, PLA in multiplekserskih vezjih pa povzroča tudi dolge zastoje, prav tako pa imajo te izvedbe zelo veliko povezav, kar pelje k dodatnim kapacitancam ter zastojem. Zato je še najprimernejša izvedba z Booleanovimi vrati, saj uporablja manj vrat, povezave so krajše, manj tranzistorjev v zaporedni vezavi pa pomeni hitrejše preklope. Zato so Boolova vrata boljša, kadar je potrebna velika učinkovitost, medtem ko ROM in PLA izvedba omogočata boljše izdelovanje, vzdrževanje, preverljivost in testiranje.

Proces sinteze pri kombinacijskih funkcijah je sestavljen iz več stopenj (pri ROM, PLA in multiplekserskem vezju sta potrebni samo prvi dve stopnji).

Prevajanje pretvori vhodni opis v standardno obliko, ki jo uporabljajo algoritmi za minimizacijo in optimizacijo. Minimizacija zmanjša čas ter število spremenljivk v tem času, kar ima za posledico manj potrebnih tranzistorjev za izvedbo. Tehnološko razporejanje zamenja Boolove operatorje z Booovimi vrati iz dane knjižnice. Optimizacija še dodatno uskladi vrata, tako da odstrani kritične poti. Pri tem se mora odločiti med velikostjo površine ter zastoji, ki sta med seboj nasprotujoči si zahtevi. Končno se definirajo še velikosti tranzistorjev, kar je potrebno za celotno učinkovitost vezja. Posamezni imajo lahko različno velikost, s tem pa tudi kapacitanco ter preklopni čas.

Končni avtomat (FSM - Finite State Machine): je najbolj uporaben model v računalniški znanosti in tehniki. Sestavljen je iz niza stanj, niza prehodov med stanji in niza dejanj v vsakem stanju ter prehodu. FSM lahko temeljijo na stanjih ali pa prehodih. Pri tistih, ki temeljijo na stanjih (Moore), je izhodna vrednost odvisna samo od stanja, pri prehodnih (Milley) pa je izhodna vrednost odvisna od stanja ter vhodnih vrednosti. Pri prehodnih FSM se izhodna vrednost spremeni, ko se spremeni vhodna vrednost, stanje pa se spremeni ob naslednjem urinem impulzu. Pri stanjskih FSM pa se izhodna vrednost ne spremeni, dokler se ne glede na spremembo vhodne vrednosti ne spremeni stanje.

Za obe vrsti avtomatov velja naslednji potek sinteze. Prevajanje pretvori vhodni opis v standardni FSM opis. Med minimizacijo stanj se izločijo vsa redundantna stanja (tista, katerih izhodi so enaki). Cilj kodiranja stanj je čim manjše število bitov, potrebnih za kodiranje posameznih stanj, s čimer zmanjšamo število flip-flopov, potrebnih za hranjenje kode stanja. Prav tako pa se, v zadnji fazi, zmanjša število funkcij naslednjih stanj in izhodnih funkcij.

Končni avtomat s podatkovnim vodilom (data path) - FSMD: za modele z veliko stanji je uporabnejši FSM s podatkovnim vodilom. To je izvedeno z uporabo spremenljivk, spravljenih v registrih in pomnilniku. Vsaka spremenljivka nadomešča veliko različnih stanj.

Sinteza se prične s prevajanjem, ki pretvori osnovni opis (HDL) v tabelarično obliko za naslednja stanja ter izhodne funkcije. Izbira enot določi število in tip

registrskih, funkcijskih ter spojnih enot. Povezovanje registrov dodeli spremenljivke posameznim registrom, povezovanje enot pa dodeli operacije funkcijskim enotam. Povezovanje spojev dodeli spojne enote podatkovnim vodilom med posameznimi enotami. Med definiranjem krmiljenja se izdelajo Boolovi izrazi za vsak signal podatkovnega vodila, ki krmili selektorje, registre in funkcijske enote. Sinteza krmiljenja ter funkcijskih enot pa končno izdelata podroben opis krmilne enote ter podatkovnega vodila.

Sistemski opis: sistem je opisan kot niz procesov. Vsak proces je lahko opisan s pomočjo zank, odločitvenih stavkov in prireditvenih stavkov. Sistemski opis predvideva obstoj registrov za globalne in lokalne spremenljivke. Prav tako predvideva, da operacije branja in hranjenja vrednosti ter prenos vrednosti iz registrov do funkcijskih enot in nazaj ne vzamejo nič časa.

Tudi sistemska sinteza se prične s pretvorbo funkcijskega opisa, napisanega v enem izmed standardnih programskih jezikov, v graf toka podatkov, ki eksplicitno prikazuje odvisnost podatkov od vhodnih opisov. Razporejanje razdeli ta prikaz v lokalne skupine, kajti vsaka skupina bo izvedena na ločenem integriranem vezju. Sinteza vmesnika določa komunikacijske kanale, protokole ter opremo za izvedbo protokolov. Razvrščanje razdeli graf toka podatkov v stanja oziroma krmilne korake. Dodelitve spremenljivk v posameznem koraku so izvedene sočasno. Tako razvrščen funkcijski opis je nato sintetiziran kot FSMD.

3 Testiranje integriranih vezij

Današnja integrirana vezja so velika, hitra, gosta in energijsko potratnejša kot kadarkoli prej. Zaradi tega je njihovo testiranje težje in dražje. Kljub natančnim načrtovalnim postopkom pa je testiranje zanesljivih, kakovostnih vezij potrebno.

Večja integrirana vezja zahtevajo več testov in jih je težje učinkovito testirati. Hitrejša vezja ne morejo biti vedno testirana z obstoječimi testnimi orodji; gostejša vezja so bolj dojemljiva za posamezne okvare in občutljivejša za napake, ki jih sedanje testne tehnike ne morejo odkriti; energijsko potratna vezja pa so prisilila k zmanjšanju napetostnih nivojev, kar pa hkrati vodi k večji občutljivosti za manjše okvare [6].

Zaradi tržnih povpraševanj pa je testiranje zelo pomembno. Zato je dobro, če ima integrirano vezje neko dodatno vezje, ki olajša testiranje samega vezja. Po drugi strani pa dodatno vezje običajno zahteva dodatno površino silicija, daljši čas izdelave, večje stroške izdelave in manjše število izdelkov v časovni enoti. Poleg tega je potrebno upoštevati še izdelavo testirne opreme in s tem povezane stroške.

3.1 Hkratno testiranje

Mnoga integrirana vezja so testirana samo s tovarniškimi testi. Uporaba vgrajenih samotestov je olajšala preizkušanje proizvodov, prav tako pa je omogočila preizkušanje samih vezij na mestu delovanja, ob vklopu ali pa celo na

oddaljenih mestih preko telefona. Nekatere izvedbe samotestov namreč omogočajo, da poteka njegovo izvajanje sočasno z delovanjem vezja. Neprestano preverjanje delovanja vezja je zlasti potrebno v primerih, kot so telefonske centrale, naprave zračnega vodenja itd. Hkratno testiranje omogoča takojšnje opozarjanje, če se v sistemu pojavi napaka. Tako se lahko na primer spremeni ali pa ustavi določen proces, če se ugotovi, da je prišlo v delovanju do napake [7].

V opisanem primeru je sočasno vgrajeno samotestiranje vključeno v proces sinteze podatkovnih poti (data-path). Testiranje vezja je časovno usklajeno s procesiranjem podatkov. To pa je tudi ena izmed prednosti glede na teste, ki potrebujejo čas samo zase, prav tako pa deluje test z enako hitrostjo, kot se procesirajo podatki in ne, kakor pri nekaterih testih, pri nižjih hitrostih.

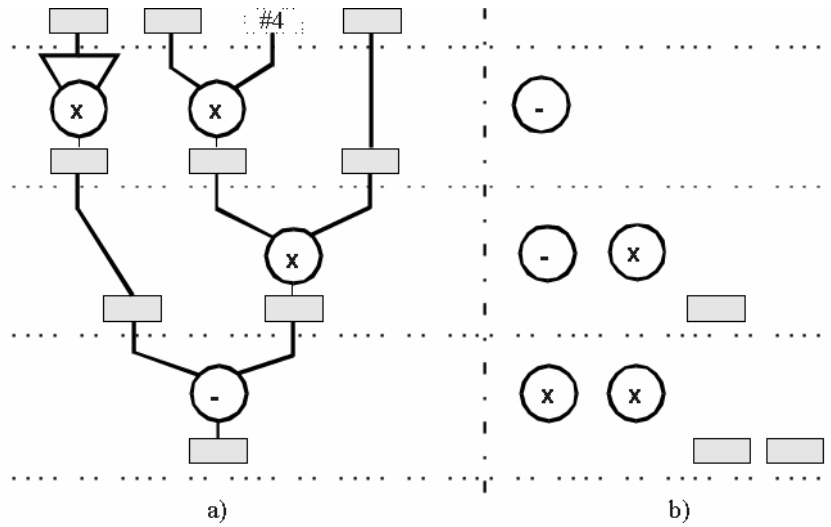
Testna metoda uporablja prevdonaključni testni vektor v vezju. V izogib velikemu številu primerjalnih vektorjev, je izvedena oznaka, neke vrste spremenljivka, ki označuje pojavitev napake. Ker je oznaka narejena hkrati s testnim vektorjem, lahko opozarja, če se je pojavila napaka, ki bi lahko vplivala na podatke.

Ta metoda preverja v glavnem delovanje podatkovnih poti in ne krmilne enote. Preizkuša samo pravilno zaporedje izvajanja ukazov ter povezave med krmilno enoto in krmiljenim elementom; ob tem pa ne preizkuša krmilnih bitov, ki, za razliko od testnih podatkov, krmilijo dejanske podatke.

3.2 Testiranje nedejavnih funkcijskih enot

V prvem koraku sinteze je algoritem delovanja vezja preveden v graf toka podatkov (data-flow). Graf toka podatkov kaže vzročnost operacij. Nato se posamezne operacije dodelijo določenemu urinemu ciklu ali krmilnemu koraku (control-step) ter določeni funkcijski enoti (množilnik, seštevalnik,...). Dodeljeni so tudi registri, ki hranijo podatke med posameznimi urinimi cikli. Vezje je popolnoma sinhrono, saj je urin signal povezan na vse sekvenčne elemente vezja. Prav tako pa mora imeti vezje tudi krmilnik, ki vodi njegovo celotno delovanje. Iz

grafa toka podatkov je razvidno, katere funkcijske enote so uporabljene v posameznem urinem ciklu (slika 3.1). Tako zlahka določimo, kdaj je posamezna funkcijska enota ali register razpoložljiv za hkratno testiranje.



Slika 3.1:
a) graf toka podatkov za $D = b^2 - 4ac$
b) neuporabljene funkcijske enote in registri

Iz funkcijskih enot in registrov, ki v posameznih trenutkih oziroma krmilnih korakih niso uporabljeni, sestavimo nov graf toka podatkov. Na določenem mestu v tem testnem grafu se tvorijo testni vektorji, ki nato sočasno s pravimi podatki krožijo po algoritmu, vendar preko funkcijskih enot in registrov, ki bi bili takrat nedejavni, tako da je tok dejanskih podatkov nemoten.

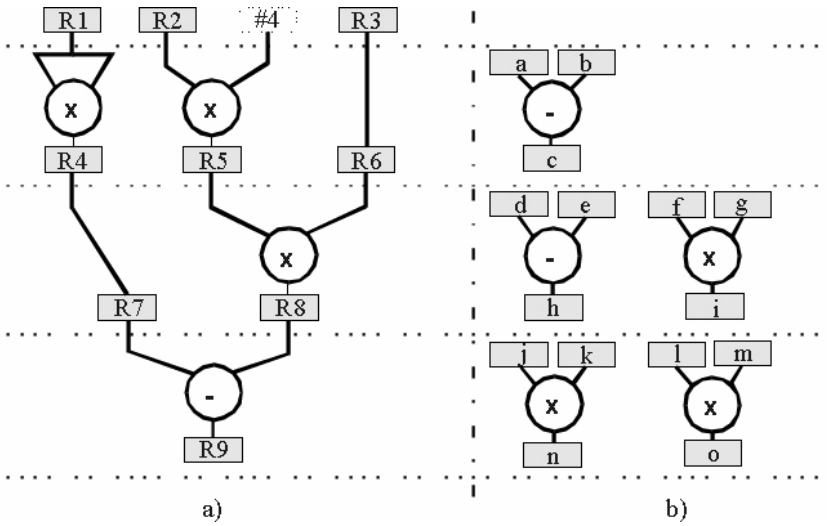
Sočasno testiranje potrebuje nekaj dodatnih registrov in multiplekserjev, ni pa potrebno povečati števila funkcijskih enot. Prav tako pa je seveda potrebno dodati psevdonaključni generator števil ter oznako, ki označuje, če se je morebiti pojavila napaka. Še posebej pri obsežnih vezjih se pozna, kako pomembno je, da ni potrebno podvajati funkcijskih enot. Edina cena, ki jo za test plačamo, je majhno povečanje števila registrov in multiplekserjev.

Poleg prednosti, pa ima tak način testiranja tudi nekaj slabosti. Nekatere funkcijske enote namreč testni vektor precej osiromašijo oziroma mu odvzamejo dobršen del teže. Tako pri množenju z 2^k dobi zadnjih k bitov vrednost nič, pri

množenju enakih vrednosti bo drugi najmanjši bit vedno 0, pri deljenju se lahko vrednost neprimerno zmanjša, pri seštevanju enakih vrednosti bo zadnji bit seštevka vedno 0,... Zaradi takih problemov je potrebno izbrati dobre testne vektorje ali pa jih po možnosti vnašati v testni algoritem na več mestih [7].

3.3 Algoritmično grajenje testnega grafa toka podatkov

Operacije v grafu toka podatkov so razvrščene v posamezne krmilne korake preden sestavimo testni graf toka podatkov. Prvotno so rezultati posameznih operacij postavljeni v ločene registre. Le-te lahko kasneje združimo ter tako zmanjšamo njihovo število. Sprva lahko tako prikažemo posamezne funkcijske enote z vhodnimi in izhodnimi registri, pri čemer se moramo zavedati, da lahko nekatere izmed vhodnih registrov kasneje združimo v en register, saj nosijo enako informacijo (slika 3.2).



Slika 3.2:
a) število različnih registrov
b) potrebni registri za vsako funkcijsko enote

3.3.1 Sestavljanje grafa s povezanimi registri

Za izgradnjo testnega grafa toka podatkov je potrebno poiskati vse prehode in jih nato preizkusiti. Obstajajo štirje tipi prehodov:

- iz registra v levi vhod funkcijske enote,

- iz registra v desni vhod funkcijske enote,

- iz funkcijske enote v register,

- iz registra v drug register.

Namesto testiranja celotne operacije, to je od vhodnega registra preko funkcijske enote in v izhodni register, testiramo raje dva ločena prehoda; prehod iz vhodnega registra v funkcijsko enoto ter prehod iz enote v izhodni register.

Graf s povezanimi registri (register-merge graph) je grafična predstavitev povezav med registri v dejanskem toku podatkov in tistimi v testnem toku podatkov. Tako povežemo registre, ki pridejo na isti vhod iste funkcijske enote.

Registre pa je potrebno (tudi zaradi zmanjšanja površine) spajati, kjer je to seveda izvedljivo (s tem namreč prihranimo pri vodilih ter multiplekserjih). Najbolj lahko izkoristimo površino, če združujemo registre na naslednji način:

- prehodi z enakim vhodom in izhodom,

- prehodi z enakim vhodom,

- prehodi z enakim izhodom,

- prehodi iz registra v register,

- prehodi med enakimi funkcijskimi enotami, kjer se vhod in izhod ravno zamenjata,

- prehodi brez skupnih značilnosti.

Navedeni vrstni red stapljanja je potreben zaradi zadovoljitve testabilnostnih zahtev.

3.3.2 Zmanjšanje grafa s povezanimi registri

Ko je graf s povezanimi registri zgrajen, ga lahko zmanjšamo. Pri zmanjševanju grafa zaporedno združujemo registre. Registri, uporabljeni v istem krmilnem koraku, ne morejo biti združeni, prav tako pa v testnem grafu ne morejo biti združeni izhodni registri, če so v istem krmilnem koraku. Čeprav lahko združimo dva vhodna registra v nekem krmilnem koraku, se temu izognemo, saj bi imela sicer funkcijska enota tako na voljo manj različnih testnih vektorjev.

Glavna značilnost združevanja registrov je zadovoljitev zahtev po testni namembnosti, druga pa je zmanjšanje števila registrov, vodil in multiplekserjev. Pri grajenju testnega grafa moramo upoštevati tako zahteve kakor tudi omejitve. Omejitve nam namreč lahko preprečijo posamezna združevanja ali pa nas prisilijo v nekatera druga združevanja.

Pri urejanju združitev se moramo držati naslednjih napotkov:

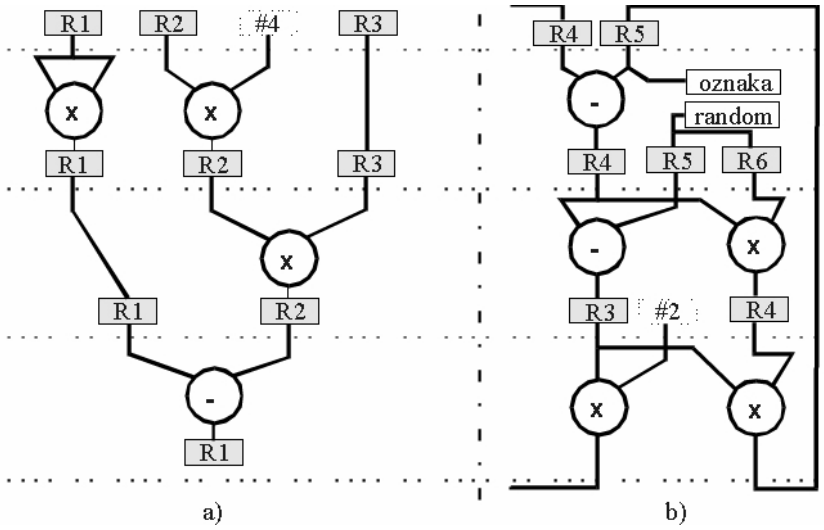
- izvedba vsake prisiljene (forced) združitve,

- razširitev vpliva prisiljenih združitev na sosednje registre, kar lahko povzroči druge prisilne združitve ali pa doda nekatere omejitve,

- če ni prisiljenih združitev, združi registre, ki zmanjšajo število multiplekserjev ter površino vodil,

- če ni drugih možnosti, izberi združitev, ki bo povzročila najmanj omejitev za ostale možne združitve,

- končno preveri vse preostale registre, ki bi še lahko pripomogli k zmanjšanju števila potrebnih registrov.



Slika 3.3:
a) povezan dejanski tok podatkov
b) povezan testni tok podatkov

Ob koncu postopka dobimo poleg osnovnega grafa toka podatkov, ki izhaja iz računskega algoritma, še graf testnega toka podatkov, ki prikazuje, kako se preračunavajo podatki v testnem algoritmu (slika 3.3).

3.4 Prednosti in slabosti metode

Na kratko bi lahko metodo predstavil z naslednjimi prednostmi:

- testiranje je izvedeno sočasno z delovanjem vezja,

- vezje se testira pri normalni hitrosti,

- ne poveča se površine zaradi uporabe velikih aritmetičnih elementov,

in slabostmi:

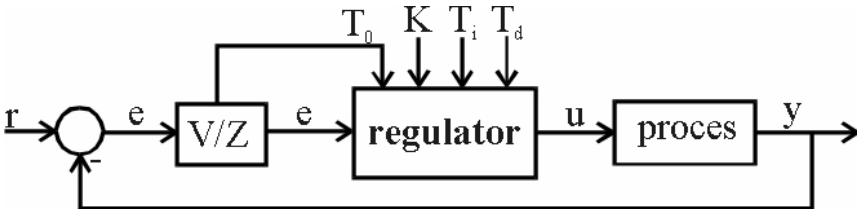
- testna pokritost v posameznih primerih ni tako velika kakor pri testnih metodah, ki postavijo testne vektorje direktno na vhod funkcijske enote, saj se za testiranje uporabi manj testnih vektorjev,

- poveča se število registrov in površina multiplekserjev ter vodil. Poveča se tudi površina krmilne enote, saj je večina krmiljenja namenjena registrom in multiplekserjem,

- metoda je odvisna od ohlapnih časov (za pretvorbe porabijo več časa, kot bi bilo dejansko potrebno) funkcijskih enot. Pri vzporednih (pipeline) sistemih so ti ohlapni časi zmanjšani, zaradi česar pa je elemente vezja težje razvrščati [7].

4 Izvedba vgrajenega samotesta

Za predstavitev reševanja danega problema sem izbral preprost algoritem diskretnega proporcionalno-integrirno-diferencirnega (PID) regulatorja. Regulator deluje tako, da iz vhodne vrednosti, ki je razlika med referenčno vrednostjo in izhodno vrednostjo procesa, izračuna vrednost regulirnega signala. Slika 4.1 prikazuje shemo postavitve regulatorja v zaprti zanki.



Slika 4.1: Regulacijska zanka

Pred regulatorjem in za njim lahko postavimo še A/D oziroma D/A pretvornik, ki poleg pretvorbe veličino po potrebi tudi skalira.

Uporabljen je algoritem, ki temelji na pravokotni metodi integracije, kjer je funkcija med dvema vzorcema predstavljena z drugim vzorcem. Funkcijo, po kateri izračunamo regulirno vrednost, zapišemo z enačbo 4.1.

$$u(k) = u(k-1) + q_0 \cdot e(k) + q_1 \cdot e(k-1) + q_2 \cdot e(k-2) \qquad (4.1)$$

Pri tem je: u(k) nova vrednost regulirne veličine, u(k-1) pretekla vrednost regulirne veličine, e(k) trenutna vrednost pogreška, e(k-1) in e(k-2) pa pretekla in predpretekla vrednost pogreška. q_0, q_1 in q_2 so koeficienti, ki jih izračunamo po enačbah 4.2, 4.3, 4.4.

$$q_0 = K(1 + \frac{T_d}{T_0}) \tag{4.2}$$

$$q_1 = -K(1 - \frac{T_0}{T_i} + 2\frac{T_d}{T_0}) \tag{4.3}$$

$$q_2 = K\frac{T_d}{T_0} \tag{4.4}$$

Oznake pomenijo naslednje: K je faktor ojačenja, T_i je časovna konstanta integrirnega dela, T_d je časovna konstanta diferencirnega dela, T_0 pa je čas vzorčenja [8].

Zaradi splošnosti regulatorja je le-ta lahko namenjen za katerikoli proces, saj dobi na vhod vrednost, ki je lahko predhodno že pretvorjena, prav tako pa je možno izhodno vrednost še pretvoriti (skaliranje, sprememba območja), preden se jo vključi v proces.

4.1 Opis vezja v VHDL in razlaga delovanja

Sam algoritem diskretnega PID regulatorja je potrebno najprej opisati v VHDL. Za to je potrebnih nekaj osnovnih ukazov, ki jih ta opisni jezik uporablja. Opis je sledeč:

```
entity pid is
port (Start      : in bit;
      NovVzorec  : in bit;
      Kp_in      : in integer;
      Ki_in      : in integer;
      Kd_in      : in integer;
      T0_in      : in integer;
      e_in       : in integer;
      uout       : out integer);
end pid;

architecture behavior of pid is
begin
   process
   variable
```

```
        Kp,Ki,Kd,T0,e,e_1,e_2,u,u_1,Temp,Temp1,Temp2,Temp3:
        integer;
    begin
        e_2 := 0;
        e_1 := 0;
        u_1 := 0;
    wait until (Start = '1');
    while (Start = '1') loop
        Kp := Kp_in;
        Kd := Kd_in;
        Ki := Ki_in;
        T0 := T0_in;
        e  := e_in;
    wait until (NovVzorec = '1');

        Temp1 := Kd / T0;
        Temp2 := Ki * T0;

        Temp  := Temp1 + Kp;
        Temp  := Temp * e;
        u     := u_1 + Temp;

        Temp3 := Temp1 * e_2;
        u     := u + Temp3;

        Temp  := 2 * Temp1;
        Temp  := Kp + Temp;
        e_2   := e_1;
        Temp  := Temp - Temp2;
        Temp  := Temp * e_1;
        u     := u-Temp;

        uout  <= u;
        e_1   := e;
        u_1   := u;

        end loop;
    end process;
end behavior;
```

Zaradi poenostavitve so v algoritmu glede na prej omenjene enačbe uporabljeni drugačni vhodi. Pri tem veljajo enačbe 4.5, 4.6 in 4.7.

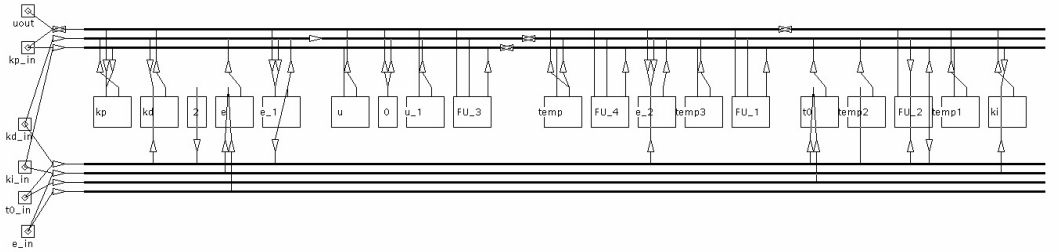
$$K_p = K \tag{4.5}$$

$$K_i = \frac{K}{T_i} \tag{4.6}$$

$$K_d = K \cdot T_d \tag{4.7}$$

Poenostavitve so potrebne zaradi splošnosti algoritma, saj se v enačbah ne pojavlja več vrednost 1. Tako lahko uporabim različna območja vrednosti, kjer so decimalne vrednosti predstavljene z izbranim število cifer celega števila (na primer 1000 predstavlja vrednost 1, 1 pa predstavlja vrednost 0,001).

Algoritem počaka, da je sistem vključen in da pride na vhod nov vzorec. Spremenljivka NovVzorec je namreč povezana s členom za vzorčenje in zadrževanje in jo ta člen postavi v položaj pripravljenosti, ko so podatki že vzorčeni. Nato se izvede zgoraj navedeni izračun nove vrednosti izhoda. To se ponavlja, dokler je regulator vključen. Opisani algoritem pa je v obliki povezav med posameznimi registri izveden na način kot ga prikazuje slika 4.2.



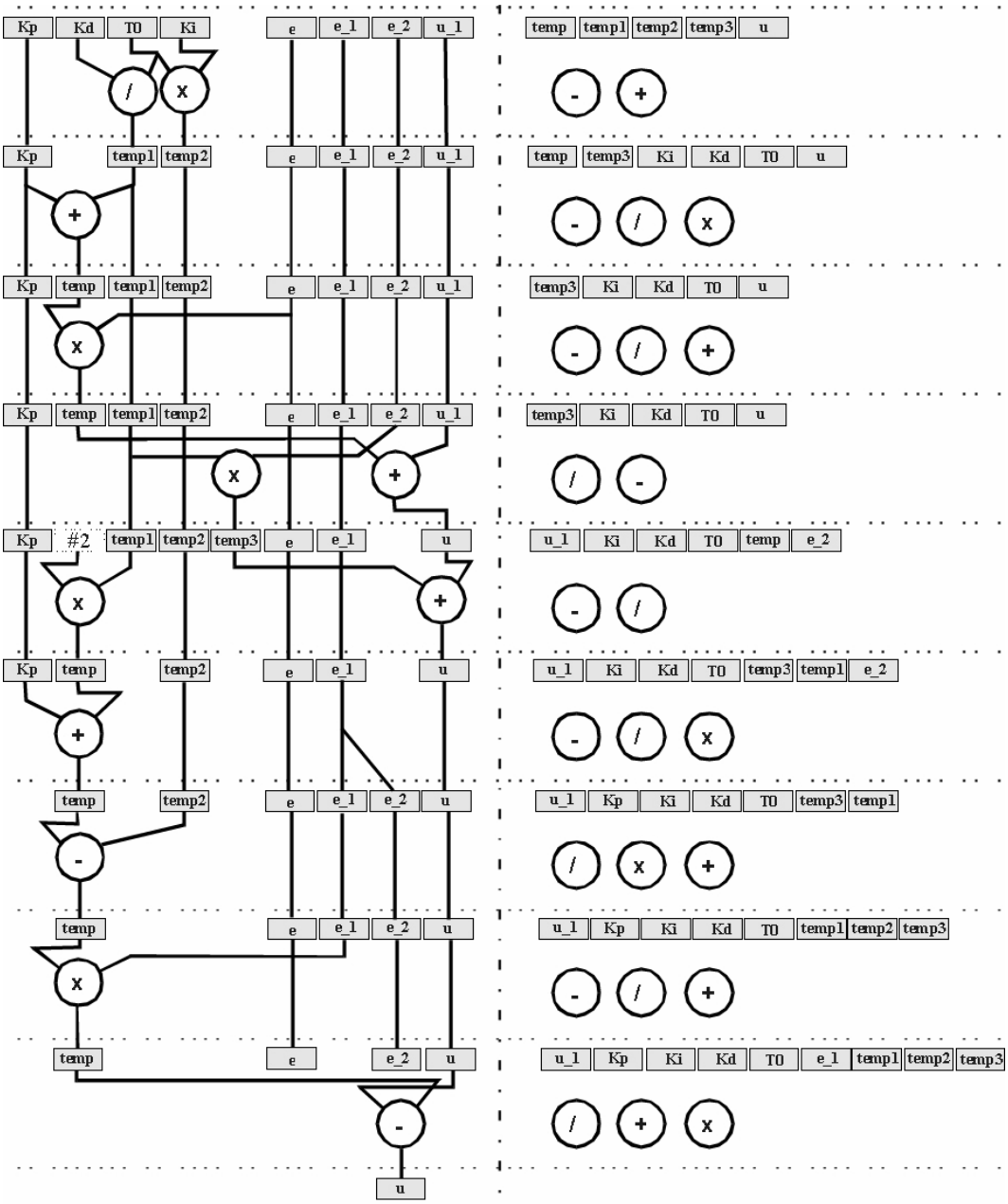
Slika 4.2: Povezave med registri, funkcijskimi enotami ter vodili

Na levi strani so predstavljeni vhodi ter izhodi iz vezja. Vodila so predstavljena z vodoravnimi črtami, ki povezujejo registre in funkcijske enote. Posamezni pravokotniki prikazujejo registre ter funkcijske enote. V registrih se shranjujejo trenutne vrednosti, v funkcijskih enotah pa se izvršijo računske operacije.

4.2 Postopek obdelave VHDL opisa

Zgoraj omenjeni algoritem lahko sedaj obdelam po že omenjenih postopkih za vgradnjo hkratnega samotesta. Najprej je potrebno sestaviti graf toka podatkov. Ta graf naredim s pomočjo programskega paketa Amical. Le-ta iz VHDL opisa s postopkom razvrščanja izdela seznam zaporedja izvajanja operacij. Vsebina datoteke *pid.mac*, ki je rezultat prevajanja in razvrščanja, je v prilogi 1.

V omenjeni datoteki so shranjeni podatki o posameznih stanjih, operacijah, ki se izvršijo v teh stanjih, ter pogojih za napredovanje v naslednje stanje. Iz teh stanj lahko razberem, kako bi izgledali posamezni krmilni koraki pri izvrševanju algoritma ter tako sestavim graf toka podatkov, ki prikazuje, kako so v posameznih krmilnih korakih uporabljeni posamezni registri in funkcijske enote. Tak graf toka podatkov prikazuje slika 4.3.



Slika 4.3:
a) potek toka podatkov
b) neuporabljene funkcijske enote in registri

Iz grafa je razvidno, da niso vse funkcijske enote ter registri uporabljeni v vseh krmilnih korakih. Iz funkcijskih enot ter registrov, ki so v posameznih korakih neuporabljeni, je torej možno sestaviti dodaten, vzporeden tok podatkov, ki bo služil testiranju funkcijskih enot. Ustrezno povezanim (neuporabljenim)

funkcijskim enotam definiram vhodne vrednosti tako, da lahko preizkusim kar največ njihovih značilnosti ter da se vrednost v testnem registru po vsaki operaciji čim bolj spremeni.

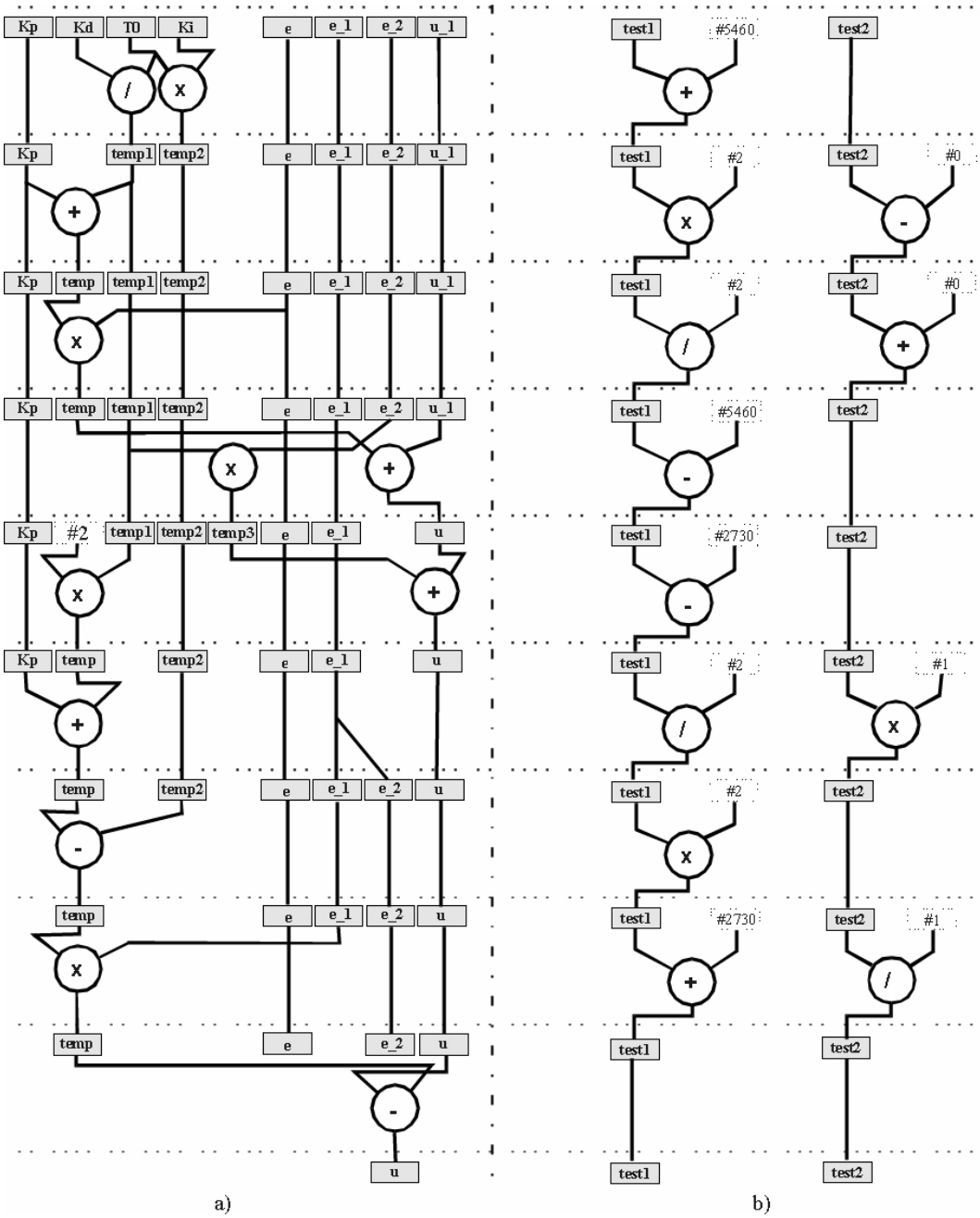
V algoritem sem moral vnesti nekaj dodatnih registrov, saj tisti, ki so bili neuporabljeni, niso povsem zadoščali za izvedbo hkratnega testiranja. Tako sem moral dodati testna registra *test1* in *test2* ter registre, ki hranijo konstantne vrednosti (1, 2730, 5460, 63495). V končnem izgledu sem tako dobil testni algoritem, ki je prikazan na desnem delu slike 4.5. Le-ta med svojim izvajanjem stestira delovanje vseh štirih funkcijskih enot. V kolikor je delovanje vezja (funkcijskih enot) pravilno, je zadnja vrednost v registrih *test1* in *test2* enaka začetni.

	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
5460	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0
+ 5460																
=10920	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0
** 2*																
=21840	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0
/ 2																
=10920	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0
- 5460																
=5460	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0
- 2730																
=2730	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0
/ 2																
=1365	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1
** 2*																
=2730	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0
+ 2730																
=5460	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	0

Slika 4.4: Vrednosti v testnem registru *test1*

V testnem registru *test1* se ves čas nahaja le nekaj (pet) vrednosti. Kljub temu pa je testiranje učinkovito, saj se med delovanjem algoritma vrednost v testnem registru stalno spreminja ter pomika v levo in desno. Tako je možno odkriti morebitno zadrgnitev posameznega biti v kakšni izmed funkcijskih enot. Algoritem je sestavljen tako, da je vsaka funkcijska enota testirana dvakrat. Na ta način se lahko odkrije napako ne glede na to ali gre za zadrgnitev na 0 ali na 1. Vrednost v registru se namreč spreminja tako, da je posamezen bit ob prvem

testiranju 0 ob drugem pa 1 ter obratno (slika 4.4). Morebitna napaka se torej širi vzdolž algoritma že od prvega ali pa šele od drugega testiranja funkcijske enote.



Slika 4.5:
a) graf toka dejanskih podatkov
b) testni del toka podatkov

Drugi testni register *test2* je dodan zato, ker prvi ne more odkriti vseh zadrgnitev na posameznih bitih. Problematični so predvsem skrajni biti pri

posameznih funkcijskih enotah, ki jih prikazuje tabela 4.1. Zaradi premikanja registra *test1* levo in desno mora imeti le-ta skrajne vrednosti vedno enake nič. S testnim registrom test2 pa se preveri tudi morebitne zadrgnitve na 0 pri teh problematičnih bitih. V drugem testnem registru se mora ob pravilnem delovanju vseh funkcijskih enot ves čas nahajati le ena vrednost (63495, ki ustreza binarni vrednosti `1111100000000111`). Z obema testnima registroma skupaj je tako zagotovljeno nadzorovanje vseh zadrgnitev na 0 in 1 na vseh bitih v vseh funkcijskih enotah.

funkcijska enota	bit z neodkrito zadrgnitvijo na 0
seštevanje	0, 1, 14, 15
odštevanje	0, 13, 14, 15
množenje	0, 2, 13, 15
deljenje	1, 12, 14, 15

Tabela 4.1: Biti z neodkritimi zadrgnitvami pri registru *test1*

Da bi se prepričal, ali bo algoritem dejansko potekal, kakor sem si zamislil, ter da bo ostal prvotni algoritem nespremenjen in nemoten, sem moral ponovno z uporabo programa za razvrščanje ugotoviti, katere funkcije se izvršujejo v posameznih korakih. Na ta način sem dobil datoteko *tpid.mac*, katere vsebina je v prilogi 2.

Očitno je, da tečeta oba toka sočasno, vendar ob tem prvotni tok nikakor ni moten, niti se podatki med seboj ne mešajo. V testni algoritem pa je potrebno dodati še spremenljivko, ki daje znak, ali je z izračunom vse v redu ali pa se je morebiti nekje pojavila napaka in je zato rezultat, ki ga je izračunal pravi algoritem, neveljaven. V ta namen se vrednost testnih registrov *test1* in *test2* primerja z njegovo izhodiščno (pričakovano) vrednostjo.

4.3 Rezultati obdelave

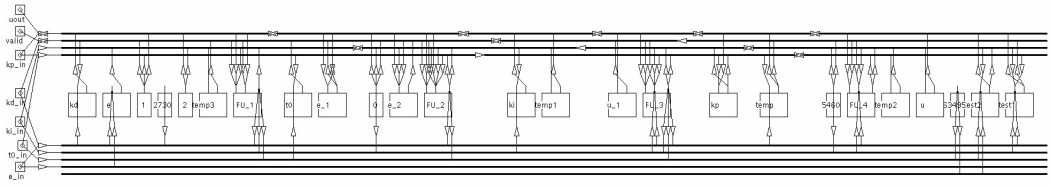
Končno pridem do naslednjega algoritma, ki se od osnovnega razlikuje v večih operacijah (število funkcijskih enot je ostalo enako) ter dodatnih registrih.

```
entity tpid is
port (Start      : in bit;
      NovVzorec  : in bit;
      Kp_in      : in integer;
      Ki_in      : in integer;
      Kd_in      : in integer;
      T0_in      : in integer;
      e_in       : in integer;
      Valid      : out bit;
      uout       : out integer);
end tpid;

architecture behavior of tpid is
begin
   process
   variable
      Kp,Ki,Kd,T0,e,e_1,e_2,u,u_1,Temp,Temp1,Temp2,Temp3,
      Test1,Test2: integer;
   begin
      e_2 := 0;
      e_1 := 0;
      u_1 := 0;
   wait until (Start = '1');
   while (Start = '1') loop
      Kp := Kp_in;
      Kd := Kd_in;
      Ki := Ki_in;
      T0 := T0_in;
      e  := e_in;
         Test1 := 5460;
         Test2 := 63495;
   wait until (NovVzorec = '1');

      Temp1 := Kd/T0;
      Temp2 := Ki*T0;
         Test1 := Test1 + 5460;

      Temp  := Temp1 + Kp;
         Test1 := Test1 * 2;
         Test2 := Test2 - 0;

      Temp  := Temp * e;
         Test1 := Test1 / 2;
         Test2 := Test2 + 0;

      u     := u_1 + Temp;
      Temp3 := Temp1 * e_2;
         Test1 := Test1 - 5460;

      u     := u + Temp3;
      Temp  := 2 * Temp1;
         Test1 := Test1 - 2730;

      Temp  := Kp + Temp;
      e_2   := e_1;
         Test1 := Test1 / 2;
         Test2 := Test2 * 1;

      Temp  := Temp - Temp2;
```

```
            Test1 := Test1 * 2;

        Temp  := Temp * e_1;
            Test1 := Test1 + 2730;
            Test2 := Test2 / 1;

        u     := u-Temp;

        uout  <= u;
            if (Test1=5460 AND Test2=63495) then Valid <= '1';
            else Valid <= '0';
            end if;
        e_1   := e;
        u_1   := u;

        end loop;
    end process;
end behavior;
```

Pri obeh algoritmih sodelujejo iste funkcijske enote. Zato je v primerih, kadar je rezultat testnega dela algoritma pravilen, velika verjetnost, da je tudi rezultat, ki je bil izračunan z dejanskimi podatki, točen. To pa se prikaže tudi na izhodu preko dodatne spremenljivke, ki označuje točnost izračuna (*Valid*=’1’). Podobno pa se na izhodu izraža tudi morebitna napaka v kateri izmed funkcijskih enot (*Valid*=’0’).

Nato uporabim še ostale dele programskega paketa, ki z algoritmi dodeljevanja funkcijskih enot, registrov ter povezav, ob upoštevanju posameznih zahtev, ki jih skupaj z utežmi zanje podam v posebni datoteki, izdelajo končni izgled integriranega vezja (prikazan je na sliki 4.6).



Slika 4.6: Povezava funkcijskih enot in vodil pri vezju s testnim algoritmom

Pomen posameznih objektov na sliki je enak kot pri sliki 4.2, prav tako pa je podobna tudi razporeditev registrov ter funkcijskih enot. Bistvena razlika je v večjem številu registrov ter večjem številu notranjih vodil, kar je posledica izvajanj več operacij hkrati.

S tem je opravljena prva faza načrtovanja integriranega vezja z vgrajenim samotestom. Zdaj je potrebno zgornji opis algoritma pretvoriti v strukturni opis.

Le-ta bo lahko potem uporabljen za dejansko izdelavo vezja. Še pomembneje pa je, da strukturni opis lahko simuliramo s simulatorjem napak, ki nam da odgovor na vprašanje, za koliko se je povečala zanesljivost delovanja vezja oziroma regulatorja, ker ima vgrajen samotest.

Sedaj z uporabo podprogramov za dodeljevanje funkcijskih enot, vodil ter povezav izdelam izhodni opis programskega paketa Amical. Ob tem so upoštevane tudi naše dodatne zahteve in omejitve, ki jih lahko posredujemo preko dodatnih, inicializacijskih datotek. Pri delu z računalnikom SUN SPARCstation 10, z operacijskim sistemom 4.1.3_U1, s procesorjem tipa sun4, z 48 Mb spomina ter z 226 Mb virtualnega (navideznega) spomina, sem potreboval za celotno sintezo, od branja funkcijskega opisa do izdelave izhodnega opisa, približno 40 minut. Večji del časa (30 minut) je program porabil za dodeljevanje funkcijskih enot, vodil ter povezav.

Izhodne datoteke, opisane v formatu solar, pa nato še prevedem s priloženim prevajalnikom v standarden opisni jezik VHDL, s katerim dela tudi večina simulatorjev napak. Za to je potrebno imeti tudi VHDL knjižnico z opisi posameznih funkcijskih enot. Celoten potek dela s programskim paketom Amical je grafično prikazan v prilogi 3.

4.4 Vrednotenje dobljenega vezja

Razpoložljivi simulatorji napak, ki so dostopni za izobraževalne namene, so zaradi svoje preprostosti neuporabni. Le-ti namreč potrebujejo bodisi kakšen specifičen format za opis vezja, ki ga programski paket Amical ne zna narediti ali pa podpirajo zgolj majhen del celotnega nabora VHDL ukazov. Pretvorba v ustreznejši format bi presegla obseg tega dela, lahko pa je cilj kakšnega drugega podobnega dela. Zaradi omenjenih težav sem bil prisiljen izvesti simulacijo napak ročno. To sem naredil tako, da sem v funkcijski opis vnesel namerne napake, simuliral delovanje vezja in opazoval, kakšne vrednosti se pojavijo na izhodu. Simulacijo sem izvedel pri več različnih vhodnih vrednostih.

Sprva sem preveril, kako zanesljivo in natančno je delovanje funkcijskih enot, ki sem jih uporabil za izvedbo seštevanja, odštevanja, množenja in deljenja. V ta namen sem v funkcijskem opisu, na mestih po izvršitvi posamezne operacije v rezultatu le-te namerno postavil enega izmed bitov na 0 oziroma 1. To sem naredil vsakič na vseh mestih v algoritmu, kjer se posamezna operacija izvrši. Tako sem dosegel učinek, ki bi ga dajala pokvarjena funkcijska enota.

Izvedel sem nekaj poskusov in prišel do naslednjih ugotovitev. Pri osnovnem vezju se v nadaljnjem postopku upoštevajo vse vrednosti, ki jih vezje pošlje v proces ne glede na to, da so izračuni nepravilni. Pri vezju z vgrajenim testnim algoritmom pa se vezje na vse enojne napake, ki so posledica zadrgnitve na 0 oziroma 1, odzove z opozorilom v obliki spremenljivke *Valid*=’0’, ki pove procesu, da rezultata ni varno uporabljati.

Dejansko sem, pri vsaki funkcijski enoti, simuliral samo zadrgnitev na 0 in 1 na samo dveh bitih, ki sem ju naključno izbral (v sredini ter na robu 16 bitnega bajta). Vseh možnih napak v vezju pa je, če upoštevamo zadrgnitve na 1 in 0 na vsakem bitu na izhodu vsake funkcijske enote, kar 128. Pri osnovnem vezju se vse te napake torej prenesejo na izhod ter naprej v proces, medtem ko jih je vezje s samotestom sposobno odkriti.

Iz tabele 4.1 je razvidno, da je dobljeno integrirano vezje zanesljivejše, saj bi se ob uporabi samo testnega registra *test1* v nadaljnjo obdelavo izmuznilo le 12,5% vrednosti, ki so zanesljivo napačne, medtem ko je ta odstotek pri osnovnem vezju kar 100%. Z drugim testnim registrom pa dosežemo, da v nadaljnjo uporabo ne gre nobena napačna vrednost. To pomeni, da je načrtano vezje glede na napake na izhodu funkcijskih enot 100% zanesljivo pri odkrivanju enojnih napak na posameznih bitih. Pri osnovnem vezju se namreč uporabijo vse izhodne vrednosti, medtem ko so v vezju s testnim algoritmom izhodne vrednosti še dodatno ovrednotene, tako da so napačne vrednosti izločene.

Glede zanesljivosti vezja zaradi uporabe posameznih registrov, kjer pride lahko do podobnih napak, pa sem ugotovil naslednje. Večine registrov se med delovanjem ne more preveriti, ker dobijo vrednosti direktno iz vhodov v

integrirano vezje. Rešitev bi bila v podvojitvi registrov, ki sprejmejo vhodno vrednost, vendar bi to preveč vplivalo na velikost vezja. Preostale registre pa sem na enak način pokvaril ter preveril odziv vezja. Ob pojavu napake v registru *temp* se le-te ne da odkriti, saj bi bilo za to potrebno bistveno spremeniti testni algoritem, hkrati pa bi se verjetno podaljšalo izvajanje osnovnega algoritma. Če pride do napake v katerem izmed testnih registrov (*test1* ali *test2*), pa ta napaka ne zmanjša zanesljivosti delovanja vezja (povzroči, da se postavi signal *Valid*=’0’). Taka napaka, ki pomeni, da je testno vezje pokvarjeno, pa se obravnava enako, kakor če bi prišlo do napake v funkcijski enoti, namreč vezje je potrebno zamenjati.

Zaradi vseh naštetih lastnosti integriranega vezja z vgrajenim testom lahko sklepam, da omenjeno vezje poveča zanesljivost, natančnost ter varnost delovanja sistema, katerega del je.

5 Sklep

5.1 Značilnosti dobljenega vezja

Vezje, ki sem ga dobil v opisanem postopku, se od osnovnega precej razlikuje. Tipične razlike prikazuje tabela 5.1.

	osnovno vezje	vezje s testnim algoritmom
registri	15	21
vodila	7	9
funkcijske enote	4	4
vhodi	5	5
izhodi	1	2

Tabela 5.1: Primerjava osnovnega vezja z načrtanim vezjem

Večje število registrov je posledica dodatnih testnih registrov ter registrov, ki hranijo vrednosti 1, 2730, 5460 in 63495. Povečanje števila notranjih vodil je posledica več sočasnih operacij. Število funkcijskih enot je enako, kar je bistveno, saj želimo preverjati njihovo delovanje. Dodaten izhod ovrednoti izračunano vrednost. Določenega povečanja je deležen tudi krmilnik, ki vodi delovanje celotnega vezja. Dobljeno vezje torej zavzame večjo površino, vendar pa se oba algoritma izvajata enako dolgo.

Poleg fizičnih lastnosti pa je pomembna tudi zanesljivost, ki govori v prid vezja z vgrajenim testnim algoritmom, saj je njegovo vezje precej bolj zanesljivo od delovanja osnovnega vezja.

5.2 Dodatne možnosti načrtovanja

Predstavljeno izvajanje izdelave samotestabilnega vezja je možno izboljšati na več načinov, kar je lahko smernica za nadaljnje delo na tem področju.

Lahko bi spremenili tok testnega algoritma, ter tako vključili v testiranje še preostale registre. V uporabi programskega paketa Amical bi lahko izrabili tudi nekatere druge možnosti, ki nam jih program ponuja: ročno izvajanje razvrščanja in dodeljevanja ali pa drugačne zahteve glede velikosti, razvrstitev in števila funkcijskih enot ter vodil.

5.3 Prednosti uporabljene metode

Za testiranje vezja bi lahko uporabil več različnih načinov. Obstaja namreč nekaj izvedb generatorjev testnih vektorjev. Le-ti so lahko psevdonaključni in obširni (exhaustive) ali pa deterministični [6].

Psevdonaključni generatorji testnih vektorjev so sicer najbolj razširjeni, vendar je njihova bistvena slabost, da mnoga vezja vsebujejo napake, ki jih odkrijejo samo nekateri vhodni vektorji, zato lahko traja kar nekaj časa, preden se taka napaka z ustreznim testnim vektorjem odkrije. Čas od nastopa napake do njenega odkritja pa je v nekaterih sistemih izredno pomemben.

Obširni generatorji so podobni psevdonaključnim, le da ti postavljajo na vhod vezja čisto vse možne kombinacije vhodnih vektorjev (tudi kombinacijo s samimi ničlami). Namenjeni so popolnemu testiranju vezij in so razen za povsem majhna vezja neprimerni in nepraktični.

Deterministični generatorji generirajo poljubno vnaprej določeno testno sekvenco. Realizirani so lahko s psevdonaključnim generatorjev, ki mu vsilimo

določeno vsebino. Lahko pa imajo vrednosti tudi zapisane v ROMu. Slabost takega generatorja je sorazmerno velika kompleksnost.

Zaradi vseh zgoraj naštetih slabosti metod sem se odločil za izvedbo z neke vrste vgrajenimi testnimi vektorji, saj tako ne potrebujem dodatnih generatorjev, odziv na napako pa je zelo hiter.

5.4 Pridobitve z uporabljenim načinom testiranja

Testiranje je eden glavnih prispevkov pri obračunavanju stroškov proizvodnje in vzdrževanja digitalnih integriranih vezij. Pomembno je zmanjšati stroške vzdrževanja (kasnejši servisi), zato je potrebno izdelati vezje, ki je samo sposobno preverjati svoje delovanje. To pa zagotovimo s primerno izbiro modelov napak ter dobrimi testnimi vektorji (oboje je dandanes že močno zastopano v načrtovalnih orodjih), saj oboje močno vpliva na uspešnost nekega vezja.

Res je, da se proizvodni stroški z načrtovanjem in vgrajevanjem dodatnega testnega vezja povečajo in s tem silijo tudi k manjši produktivnosti (manj izdelkov v časovni enoti), vendar pa takšno testiranje pripomore k zanesljivejšemu delovanju samega vezja ter boljšemu in varnejšemu poteku v celotnem sistemu, katerega del je integrirano vezje (v tem primeru gre za regulator). Integriranega vezja namreč ni potrebno dodatno testirati ter zaradi tega zaustavljati delovanja celotnega sistema. Če že pride do okvare na njem, nam to sam sporoči, kar je koristen podatek tudi za opremo, ki sicer uporablja izhodne vrednosti integriranega vezja. Sistem tako preneha uporabljati vrednosti, ki so zelo verjetno nepravilne, s čimer se izognemo tudi marsikateri dodatni okvari na ostali opremi, vključeni v sistem.

Ob vsem naštetem je pomembno tudi dejstvo, da večina proizvajalcev obsežnih integriranih vezij odobrava pokritost napak z vsaj 99%. Kljub temu, da želijo zelo zanesljive ter poceni elemente, pa cilj proizvajalcev ni samo priti do minimuma skupnih stroškov, temveč je potrebno te stroške stalno zmanjševati (tako pri izdelavi kot tudi pri izvedbi testiranja).

Seznam uporabljenih virov

[1] Z. Navabi, VHDL Analysis and Modeling of Digital Systems, McGraw Hill International Editions, Singapore, 1993.

[2] A. Biasizzo, F. Novak, N. Šutanovac, Simulacija napak v digitalnih vezjih, IJS Delovno poročilo, Ljubljana, december 1991, str. 1-11.

[3] T. Kim, Scheduling and Allocating problems in High Level Synthesis, Ph.D. Thesis, University of Illinois at Urbana-Champaign, 1993.

[4] D. Gajski, N. Dutt, A. Wu, S. Lin, High-Level Synthesis: Introduction to Chip and System Design, Kluwer Academic Publishers, Dordrecht, 1992.

[5] J. Šilc, Scheduling Strategies in High Level Synthesis, Informatica 18, 1994, str. 71-79

[6] B.T. Murray, J. P. Hayes, Testing ICs: Getting to the Core of the Problem, Computer, november 1996, str. 32-38.

[7] R. Singh, J. Knight, Concurrent Testing in High Level Synthesis, Proc. Seventh International Symposium on High Level Synthesis, Niagara-on-the-Lake, Ontario, Canada, Maj 1994, str. 96-103.

[8] D. Matko, Računalniško vodenje procesov, Fakulteta za elektrotehniko in računalništvo, Ljubljana, 1995, str. 96-98.

Priloge

Priloga 1: Datoteka PID.MAC:

```
(DESCRIPTION pid
 -- Pipe depth :  1
(EXTERNAL_BUS start  (WIDTH 0  1) (TYPE BIT )(DIRECTION  IN))
(EXTERNAL_BUS novvzorec  (WIDTH 0  1) (TYPE BIT )(DIRECTION  IN))
(EXTERNAL_BUS kp_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS ki_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS kd_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS t0_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS e_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS uout  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  OUT))
(VARIABLE kp (WIDTH 0  16))
(VARIABLE ki (WIDTH 0  16))
(VARIABLE kd (WIDTH 0  16))
(VARIABLE t0 (WIDTH 0  16))
(VARIABLE e (WIDTH 0  16))
(VARIABLE e_1 (WIDTH 0  16))
(VARIABLE e_2 (WIDTH 0  16))
(VARIABLE u (WIDTH 0  16))
(VARIABLE u_1 (WIDTH 0  16))
(VARIABLE temp (WIDTH 0  16))
(VARIABLE temp1 (WIDTH 0  16))
(VARIABLE temp2 (WIDTH 0  16))
(VARIABLE temp3 (WIDTH 0  16))

(MACRO_CYCLE 1
       (STATE S1)
       (NEXTSTATE S2)
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 17 ))
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 18 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 19 ))
)

(MACRO_CYCLE 2
       (STATE S2)
       (CONDITION  (& (= ( start 1))(= ( start 1))))
       (NEXTSTATE S3)
(TRANSFER (OUTPUT kp) (INPUT kp_in )(LINE 23 ))
(TRANSFER (OUTPUT kd) (INPUT kd_in )(LINE 24 ))
(TRANSFER (OUTPUT ki) (INPUT ki_in )(LINE 25 ))
(TRANSFER (OUTPUT t0) (INPUT t0_in )(LINE 26 ))
```

```
(TRANSFER (OUTPUT e) (INPUT e_in )(LINE 27 ))
)

(MACRO_CYCLE 3
        (STATE S2)
        (CONDITION  (& (= ( start 1))(/= ( start 1))))
        (NEXTSTATE S2)
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 17 ))
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 18 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 19 ))
)

(MACRO_CYCLE 4
        (STATE S2)
        (CONDITION  (/= ( start 1)))
        (NEXTSTATE S2)
)

(MACRO_CYCLE 5
        (STATE S3)
        (CONDITION  (= ( novvzorec 1)))
        (NEXTSTATE S4)
(OPERATION / (OUTPUT temp1) (INPUT kd t0 )(LINE 30 ))
(OPERATION * (OUTPUT temp2) (INPUT ki t0 )(LINE 31 ))
)

(MACRO_CYCLE 6
        (STATE S3)
        (CONDITION  (/= ( novvzorec 1)))
        (NEXTSTATE S3)
)

(MACRO_CYCLE 7
        (STATE S4)
        (NEXTSTATE S5)
(OPERATION + (OUTPUT temp) (INPUT temp1 kp )(LINE 33 ))
)

(MACRO_CYCLE 8
        (STATE S5)
        (NEXTSTATE S6)
(OPERATION * (OUTPUT temp) (INPUT temp e )(LINE 35 ))
)

(MACRO_CYCLE 9
        (STATE S6)
        (NEXTSTATE S7)
(OPERATION + (OUTPUT u) (INPUT u_1 temp )(LINE 37 ))
(OPERATION * (OUTPUT temp3) (INPUT temp1 e_2 )(LINE 38 ))
)

(MACRO_CYCLE 10
        (STATE S7)
        (NEXTSTATE S8)
(OPERATION + (OUTPUT u) (INPUT u temp3 )(LINE 40 ))
(OPERATION * (OUTPUT temp) (INPUT 2 temp1 )(LINE 41 ))
)

(MACRO_CYCLE 11
        (STATE S8)
        (NEXTSTATE S9)
(OPERATION + (OUTPUT temp) (INPUT kp temp )(LINE 43 ))
(TRANSFER (OUTPUT e_2) (INPUT e_1 )(LINE 44 ))
)

(MACRO_CYCLE 12
```

```
        (STATE S9)
        (NEXTSTATE S10)
(OPERATION - (OUTPUT temp) (INPUT temp temp2 )(LINE 46 ))
)

(MACRO_CYCLE 13
        (STATE S10)
        (NEXTSTATE S11)
(OPERATION * (OUTPUT temp) (INPUT temp e_1 )(LINE 48 ))
)

(MACRO_CYCLE 14
        (STATE S11)
        (NEXTSTATE S12)
(OPERATION - (OUTPUT u) (INPUT u temp )(LINE 50 ))
)

(MACRO_CYCLE 15
        (STATE S12)
        (CONDITION  (= ( start 1)))
        (NEXTSTATE S3)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 52 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 53 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 54 ))
(TRANSFER (OUTPUT kp) (INPUT kp_in )(LINE 23 ))
(TRANSFER (OUTPUT kd) (INPUT kd_in )(LINE 24 ))
(TRANSFER (OUTPUT ki) (INPUT ki_in )(LINE 25 ))
(TRANSFER (OUTPUT t0) (INPUT t0_in )(LINE 26 ))
(TRANSFER (OUTPUT e) (INPUT e_in )(LINE 27 ))
)

(MACRO_CYCLE 16
        (STATE S12)
        (CONDITION  (/= ( start 1)))
        (NEXTSTATE S13)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 52 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 53 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 54 ))
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 17 ))
)

(MACRO_CYCLE 17
        (STATE S13)
        (NEXTSTATE S2)
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 18 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 19 ))
)
)
```

Priloga 2: Datoteka TPID.MAC:

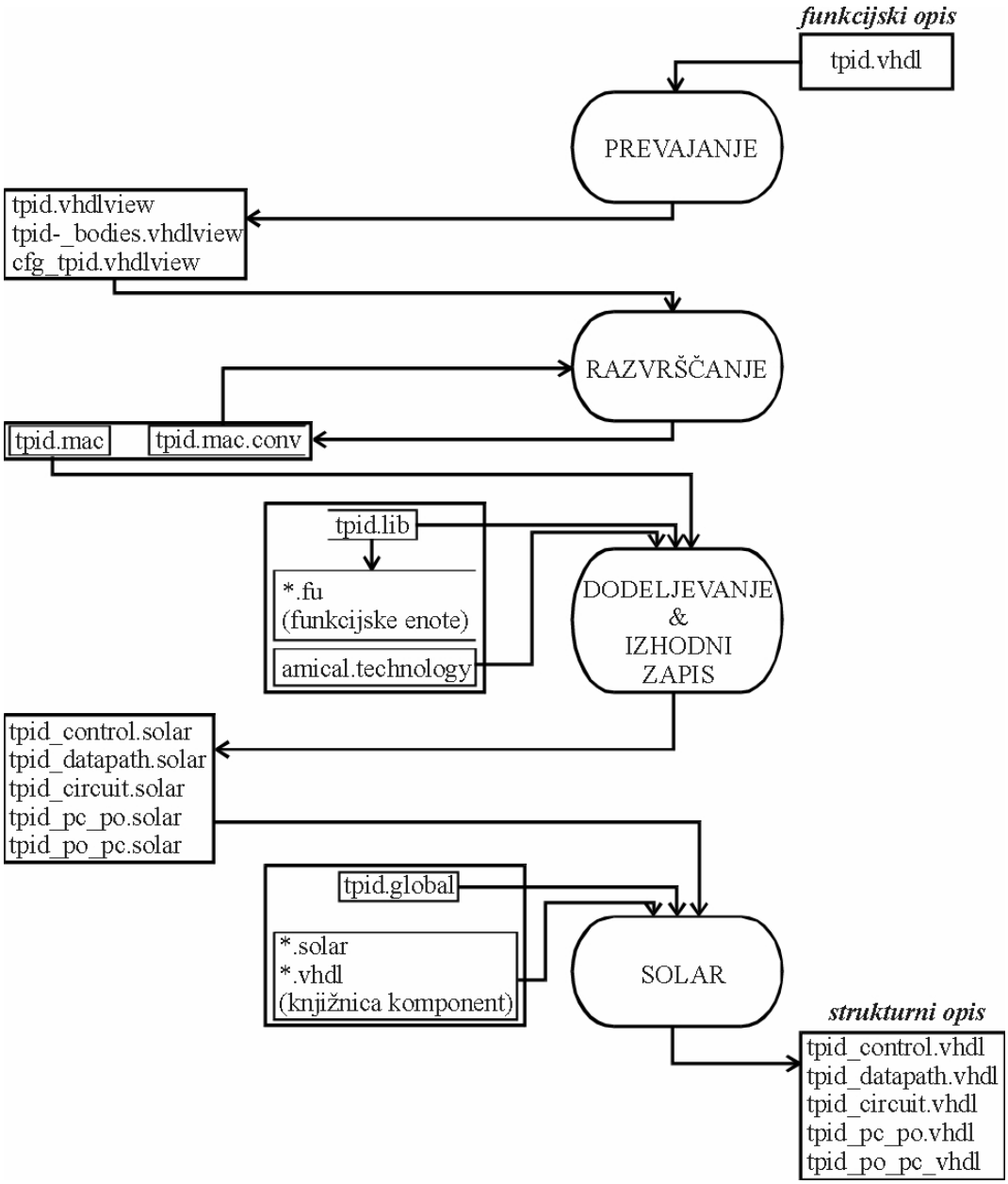
```
(DESCRIPTION tpid
 -- Pipe depth :  1
(EXTERNAL_BUS start  (WIDTH 0  1) (TYPE BIT )(DIRECTION  IN))
(EXTERNAL_BUS novvzorec  (WIDTH 0  1) (TYPE BIT )(DIRECTION  IN))
(EXTERNAL_BUS kp_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS ki_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS kd_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS t0_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS e_in  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  IN))
(EXTERNAL_BUS valid  (WIDTH 0  1) (TYPE BIT )(DIRECTION  OUT))
(EXTERNAL_BUS uout  (WIDTH 0  16) (TYPE INTEGER )(DIRECTION  OUT))
(VARIABLE kp (WIDTH 0  16))
```

```
(VARIABLE ki (WIDTH 0  16))
(VARIABLE kd (WIDTH 0  16))
(VARIABLE t0 (WIDTH 0  16))
(VARIABLE e (WIDTH 0  16))
(VARIABLE e_1 (WIDTH 0  16))
(VARIABLE e_2 (WIDTH 0  16))
(VARIABLE u (WIDTH 0  16))
(VARIABLE u_1 (WIDTH 0  16))
(VARIABLE temp (WIDTH 0  16))
(VARIABLE temp1 (WIDTH 0  16))
(VARIABLE temp2 (WIDTH 0  16))
(VARIABLE temp3 (WIDTH 0  16))
(VARIABLE test1 (WIDTH 0  16))
(VARIABLE test2 (WIDTH 0  16))


(MACRO_CYCLE 1
       (STATE S1)
       (NEXTSTATE S2)
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 18 ))
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 19 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 20 ))
)

(MACRO_CYCLE 2
       (STATE S2)
       (CONDITION  (& (= ( start 1))(= ( start 1))))
       (NEXTSTATE S3)
(TRANSFER (OUTPUT kp) (INPUT kp_in )(LINE 24 ))
(TRANSFER (OUTPUT kd) (INPUT kd_in )(LINE 25 ))
(TRANSFER (OUTPUT ki) (INPUT ki_in )(LINE 26 ))
(TRANSFER (OUTPUT t0) (INPUT t0_in )(LINE 27 ))
(TRANSFER (OUTPUT e) (INPUT e_in )(LINE 28 ))
(TRANSFER (OUTPUT test1) (INPUT 5460 )(LINE 29 ))
(TRANSFER (OUTPUT test2) (INPUT 63495 )(LINE 30 ))
)

(MACRO_CYCLE 3
       (STATE S2)
       (CONDITION  (& (= ( start 1))(/= ( start 1))))
       (NEXTSTATE S2)
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 18 ))
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 19 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 20 ))
)

(MACRO_CYCLE 4
       (STATE S2)
       (CONDITION  (/= ( start 1)))
       (NEXTSTATE S2)
)

(MACRO_CYCLE 5
       (STATE S3)
       (CONDITION  (= ( novvzorec 1)))
       (NEXTSTATE S4)
(OPERATION / (OUTPUT temp1) (INPUT kd t0 )(LINE 33 ))
(OPERATION * (OUTPUT temp2) (INPUT ki t0 )(LINE 34 ))
(OPERATION + (OUTPUT test1) (INPUT test1 5460 )(LINE 35 ))
)

(MACRO_CYCLE 6
       (STATE S3)
       (CONDITION  (/= ( novvzorec 1)))
       (NEXTSTATE S3)
)
```

```
(MACRO_CYCLE 7
       (STATE S4)
       (NEXTSTATE S5)
(OPERATION + (OUTPUT temp) (INPUT temp1 kp )(LINE 37 ))
(OPERATION * (OUTPUT test1) (INPUT test1 2 )(LINE 38 ))
(OPERATION - (OUTPUT test2) (INPUT test2 0 )(LINE 39 ))
)

(MACRO_CYCLE 8
       (STATE S5)
       (NEXTSTATE S6)
(OPERATION * (OUTPUT temp) (INPUT temp e )(LINE 41 ))
(OPERATION / (OUTPUT test1) (INPUT test1 2 )(LINE 42 ))
(OPERATION + (OUTPUT test2) (INPUT test2 0 )(LINE 43 ))
)

(MACRO_CYCLE 9
       (STATE S6)
       (NEXTSTATE S7)
(OPERATION + (OUTPUT u) (INPUT u_1 temp )(LINE 45 ))
(OPERATION * (OUTPUT temp3) (INPUT temp1 e_2 )(LINE 46 ))
(OPERATION - (OUTPUT test1) (INPUT test1 5460 )(LINE 47 ))
)

(MACRO_CYCLE 10
       (STATE S7)
       (NEXTSTATE S8)
(OPERATION + (OUTPUT u) (INPUT u temp3 )(LINE 49 ))
(OPERATION * (OUTPUT temp) (INPUT 2 temp1 )(LINE 50 ))
(OPERATION - (OUTPUT test1) (INPUT test1 2730 )(LINE 51 ))
)

(MACRO_CYCLE 11
       (STATE S8)
       (NEXTSTATE S9)
(OPERATION + (OUTPUT temp) (INPUT kp temp )(LINE 53 ))
(TRANSFER (OUTPUT e_2) (INPUT e_1 )(LINE 54 ))
(OPERATION / (OUTPUT test1) (INPUT test1 2 )(LINE 55 ))
(OPERATION * (OUTPUT test2) (INPUT test2 1 )(LINE 56 ))
)

(MACRO_CYCLE 12
       (STATE S9)
       (NEXTSTATE S10)
(OPERATION - (OUTPUT temp) (INPUT temp temp2 )(LINE 58 ))
(OPERATION * (OUTPUT test1) (INPUT test1 2 )(LINE 59 ))
)

(MACRO_CYCLE 13
       (STATE S10)
       (NEXTSTATE S11)
(OPERATION * (OUTPUT temp) (INPUT temp e_1 )(LINE 61 ))
(OPERATION + (OUTPUT test1) (INPUT test1 2730 )(LINE 62 ))
(OPERATION / (OUTPUT test2) (INPUT test2 1 )(LINE 63 ))
)

(MACRO_CYCLE 14
       (STATE S11)
       (NEXTSTATE S12)
(OPERATION - (OUTPUT u) (INPUT u temp )(LINE 65 ))
)

(MACRO_CYCLE 15
       (STATE S12)
       (CONDITION (&(&(=(test1 5460))(=(test2 63495)))(=(start 1))))
```

```
        (NEXTSTATE S3)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 67 ))
(TRANSFER (OUTPUT valid) (INPUT 1 )(LINE 68 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 69 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 70 ))
(TRANSFER (OUTPUT kp) (INPUT kp_in )(LINE 24 ))
(TRANSFER (OUTPUT kd) (INPUT kd_in )(LINE 25 ))
(TRANSFER (OUTPUT ki) (INPUT ki_in )(LINE 26 ))
(TRANSFER (OUTPUT t0) (INPUT t0_in )(LINE 27 ))
(TRANSFER (OUTPUT e) (INPUT e_in )(LINE 28 ))
(TRANSFER (OUTPUT test1) (INPUT 5460 )(LINE 29 ))
(TRANSFER (OUTPUT test2) (INPUT 63495 )(LINE 30 ))
)

(MACRO_CYCLE 16
        (STATE S12)
        (CONDITION (&(&(=(test1 5460))(=(test2 63495)))(/=(start 1))))
        (NEXTSTATE S13)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 67 ))
(TRANSFER (OUTPUT valid) (INPUT 1 )(LINE 68 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 69 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 70 ))
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 18 ))
)

(MACRO_CYCLE 17
        (STATE S12)
        (CONDITION(&(|(/=(test1 5460))(/=(test2 63495)))(=(start 1))))
        (NEXTSTATE S3)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 67 ))
(TRANSFER (OUTPUT valid) (INPUT 0 )(LINE 68 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 69 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 70 ))
(TRANSFER (OUTPUT kp) (INPUT kp_in )(LINE 24 ))
(TRANSFER (OUTPUT kd) (INPUT kd_in )(LINE 25 ))
(TRANSFER (OUTPUT ki) (INPUT ki_in )(LINE 26 ))
(TRANSFER (OUTPUT t0) (INPUT t0_in )(LINE 27 ))
(TRANSFER (OUTPUT e) (INPUT e_in )(LINE 28 ))
(TRANSFER (OUTPUT test1) (INPUT 5460 )(LINE 29 ))
(TRANSFER (OUTPUT test2) (INPUT 63495 )(LINE 30 ))
)

(MACRO_CYCLE 18
        (STATE S12)
        (CONDITION(&(|(/=(test1 5460))(/=(test2 63495)))(/=(start 1))))
        (NEXTSTATE S13)
(TRANSFER (OUTPUT uout) (INPUT u )(LINE 67 ))
(TRANSFER (OUTPUT valid) (INPUT 0 )(LINE 68 ))
(TRANSFER (OUTPUT e_1) (INPUT e )(LINE 69 ))
(TRANSFER (OUTPUT u_1) (INPUT u )(LINE 70 ))
(TRANSFER (OUTPUT e_2) (INPUT 0 )(LINE 18 ))
)

(MACRO_CYCLE 19
        (STATE S13)
        (NEXTSTATE S2)
(TRANSFER (OUTPUT e_1) (INPUT 0 )(LINE 19 ))
(TRANSFER (OUTPUT u_1) (INPUT 0 )(LINE 20 ))
)
)
```

Priloga 3: Grafični prikaz načrtovanja v Amicalu



Zahvala

Zahvaljujem se:

- prof. dr. Baldomiru Zajcu za mentorstvo pri izdelavi naloge,
- asistentu mag. Andreju Žemvi za nasvete in izkazano pomoč,
- dr. Francu Novaku za koristne nasvete s področja testiranja,
- dr. Juriju Šilcu za uporabne napotke s področja visokonivojske sinteze,
- osebju oddelka za računalniške sisteme na Inštitutu Jožef Stefan za pomoč in nasvete,
- Fakulteti za elektrotehniko in Inštitutu Jožef Stefan, ker sta mi omogočila delo na njihovi računalniški opremi,
- staršem, ki so mi omogočili študij in me pri tem podpirali,
- Inštitutu Jožef Stefan, ki me je štipendiral v času študija ter mi nudil dodatno izobraževanje,
- Mirki, ki mi je ves čas pisanja tega dela stala ob strani.

Prav tako se zahvaljujem tudi vsem drugim, ki so kakorkoli pomagali pri nastanku te naloge.

Izjava

Izjavljam, da sem diplomsko delo samostojno izdelal pod vodstvom mentorja prof. dr. Baldomira Zajca. Izkazano pomoč drugih sodelavcev sem v celoti navedel v zahvali.