TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAF70LT
Gunnar Mägi 122076IASMM

DIGITAALSKEEMIDE FUNKTSIONAALSE ISETESTIMISE KVALITEEDI PARANDAMINE.

Magistritöö

Juhendajad:
Raimund Ubar
Professor / Ph.D

Sergei Kostin
teaduskraad: Doktor

Tallinn 2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

_____ _____

(kuupäev) (allkiri)

ARVUTITEHNIKA INSTITUUT
TALLINNA TEHNIKAÜLIKOOL

MAGISTRITÖÖ ÜLESANNE

Üliõpilane: Gunnar Mägi Matrikkel:122076

Lõputöö teema eesti keeles:

Digitaalskeemide funktsionaalse isetestimise kvaliteedi parandamine.

Lõputöö teema inglise keeles:

Improvement of the functional selftest quality for digital circuits

Juhendaja (nimi, töökoht, teaduslik kraad, allkiri):
Raimund Ubar, Professor, Ph.D,

Konsultandid: Sergei Kostin

Lahendatavad küsimused ning lähtetingimused:

1. Uurida meetodeid digitaalskeemide funktsionaalse isetestimise kvaliteedi tõstmiseks.
2. Töötada välja algoritm ja programm testpunktide lisamiseks.
3. Uurida võimalusi testpunktide arvu minimeerimiseks.
4. Viia läbi eksperimentide seeria kahe meetodi võrdlemiseks skeemi testitavuse parandamisel: deterministlike testide lisamine ja testpunktide lisamine.

Eritingimused:

Kasutada eksperimenteerimisel Arvutitehnika instituudi diagnostikatarkvara Turbo – Testrit ja katseskeeme ISCAS'85

Nõuded vormistamisele: Vastavalt Arvutitehnika instituudis kehtivatele nõuetele

Lõputöö estamise tähtaeg: _____

Ülesande vastu võtnud: 31 .Jaanuar . 2014. a.

_____

(lõpetaja allkiri)

kasutatud lühendite/terminite loetelu

1. ALU – Aritmeetika-loogika seade
2. ATPG – Automaatne testide generaator
3. BDD (Binary Decision Diagram) – Kahendsüsteemis otsustus diagram
4. BIST (built-in self-test) – skeemide isetestimine
5. CUT (circuit under test) – Testitav skeem
6. LFSR – Lineaarse tagasisidega nihkeregister
7. PRPG – Juhuslike algandmetega testide generaator.
8. SSBDD (Structurally Synthesized BDDs) – Struktuurselt sünteesitud otsustusdiagrammid
9. TT – „Turbo Tester“. Programmi nimi, mida sai Magistritöö raames kasutatud ja arendatud.
10. UnTested - Testi lõpuks jäänud testimata sõlmede arv.

Annotatsioon

Lõputöö on kirjutatud Eesti keeles ning sisaldab teksti 59 leheküljel, 4 peatükki, 23 joonist, 5 tabelit.

Töö eesmärgiks oli uurida meetodeid digitaalskeemide funktsionaalse isetestimise kvaliteedi tõstmiseks. Viidi läbi ulatuslik eksperimentide seeria, mille käigus võrreldi kahte meetodit: skeemi testitavuse parandamine mida täiendati töö käigus kahe uue meetodi, testpunktide skeemi lisamise teel ja testimise kvaliteedi tõstmine täiendavate deterministlike testide kasutamise abil. Eksperimentaalse uurimistöö läbiviimiseks loodi vajalik tarkvaraline keskkond, mida töö käigus täiendati kahe uue meetodi väljatöötamisega, millest üks võimaldas minimeerida vajalike testpunktide arvu ja teine võimaldas tõsta eksperimentaalse uurimistöö produktiivsust (kiirust).

Töö tulemused on vormistatud teaduslku ariklina, mis on esitatud konverentsile Baltic Electronics Conference, Oct. 4-6, 2014.

Abstract

Improvement of the functional selftest quality for digital circuits

The thesis is in Estonian language and contains 59 pages of text, 4 chapters, 23 figures, 5 tables.

Two approaches to improve the fault coverage of functional logic BIST in digital circuits are investigated and comapred. The first approach is based on introducing of additional test points. An experimental tool set is developed for fast evaluation of the number of control points that is needed to achieve 100% fault coverage for the given functional test sequence. A novel algorithm is proposed to minimize the number of control points. The second compared approach is based on complementing the functional test with additional deterministic test patterns. The pros and cons of both approaches are highlighted and discussed

The results of the master thesis are presented as a scientific article and submitted to Baltic Electronics Conference, Oct. 4-6, 2014.

Sisukord

Tabelite nimekiri

Jooniste nimekiri

1. Sissejuhatus

Mikroelektroonika ülikiire areng on võimaldamas inseneridel luua üha keerukamaid digitaalskeeme ja -süsteeme ning integreerida neid ühele ainsale kiibile. Skeemide keerukuse kasvades aga suureneb ka testimisele kuluv aeg ning maksumus. Traditsiooniliselt on testimiseks kasutatud väliseid testseadmeid ehk automaatseid testreid, millel on aga ka olulisi puuduseid. Ühest küljest halveneb skeemide keerukuse kasvades ligipääs skeemi sisepunktidele, mis on aga skeemide töö jälgimiseks ja kontrollimiseks väga oluline, ning teisest küljest halvenevad ka skeemide testimise võimalused dünaamikas, sest testrite kiiruslikud omadused jäävad kiiresti maha uutel tehnoloogiatel konstrueeritud testitavate kiipide töökiirustest. Aeglaste testritega ei ole võimalik testida üha kiiremini töötavaid mikroskeeme. Sel põhjusel on juba mõnda aega põhiliseks testimismeetodiks kujunenud digitaalsüsteemides nn. isetestimine, kus testvektoreid skeemi erinevate komponentide testimiseks genereeritakse süsteemide enda sees ning ka testimistulemuste kontroll ja analüüs toimub sammuti süsteemide enese sees [1].

Traditsiooniliselt kasutatakse digitaalsüsteemide isetestimisel nn. pseudojuhuslikku testimist, kus testitavate skeemide sisendisse saadetakse spetsiaalsete skeemide – aparatuursete testigeneraatorite poolt genereeritud pseudojuhuslikke testvektoreid ja testitavate skeemide väljundreaktsioone jälgitakse spetsiaalsete signatuuranalüsaatoritega [1]. Niisugust isetestimismeetodit nimetatakse ingl.k. terminiga BIST ehk Built-in Self-Test.

Isetestimise valdkonnas on viimasel ajal järjest rohkem päevakorrale tõusnud nn. funktsionaalse BIST-i (FBIST) meetodid, mille korral pseudojuhuslikud

testimissignaalid asendatakse testitava süsteemi enda poolt genereeritud töösignaalidega [3]. See võimaldab kokku hoida aparatuurseid kulutusi, kuna ei ole enam vaja ehitada skeemidesse sisse spetsiaalseid testvektorite generaatoreid. Teiseks eeliseks töösignaalide kasutamisel on testimise tulemuste kõrgem kvaliteet, kuna testimine toimub reaalsetes töötingimustes. Funktsionaalse isetestimise puuduseks on aga tavaliselt halvem rikete kate, kuna rikete „tabamisel“ töösignaalide abil on samuti juhuslik iseloom nagu pseudojuhuslikulgi testimisel. Sageli on see veel halvemgi, kuna töösignaalidel puudub niisugune „mitmekülgsus“, mis on omane pseudojuhuslikel signaalidel.

Olukorda, kus on raske saavutada testimisel kõrget rikete katet, nimetatakse skeemide „halvasti testitavuseks“ ja rikkeid, mida on raske tabada, nimetatakse „raskesti testitavateks riketeks“ [1].

Skeemide isetestimise parandamiseks kasutatakse kahte lähenemisviisi: hübriidset isetestimist ja skeemide testitavuse parandamist. Hübriidse isetestimise puhul ühitatakse kahe erineva testvektorite allika kasutamist [3]:

- funktsionaalsed testvektorid, mida genereerib testitav süsteem ise, ja
- täiendavad deterministlikud testvektorid, mida genereeritakse eelnevalt raskesti testitavate rikete avastamiseks ja mida hoitakse mälus.

Skeemide testitavuse parandamiseks kasutatakse samuti kahte meetodit [1, 4]:

- skeemi sisepunktide juhitavuse parandamist ja
- skeemi sisepunktide jälgitavuse parandamist.

Mõlemad süsteemide isetestimise meetodid on seotud täiendava aparatuuri kasutamisega: hübriidse isetestimise puhul on tarvilik lisamälu täiendavate deterministlike testvektorite salvestamiseks, testitavuse parandamise korral on aga vaja täiendavaid loogikaelemente skeemi juurde lisada.

Käesoleval tööl on kaks eesmärki:

- välja töötada algoritm ja programm, mis võimaldab minimeerida testpunktide arvu, mis on vajalik skeemi testitavuse tagamiseks skeemi sisesignaalide juhitavuse parandamise abil ja
- võrrelda omavahel kahte lähenemisviisi isetestimise parandamiseks – hübriidset isetestimist ja signaalide juhitavuse parandamist.

Töö teostamiseks kasutatakse rahvusvaheliselt tunnustatud ISCAS'85 perekonna katseskeeme ja arvutitehnika instituudis kasutatavaid testimise automatiseerimise töökeskkonda Turbo-Tester – rikete simuleerimiseks, deterministlike testvektorite genereerimiseks ja isetestimise kvaliteedi hindamiseks.

Nimetatud töökeskkonda sai täiendatud digitaalskeemide testitavuse parandamise programmiga, millel on kaks funktsiooni:

- arvutada vajalike testpunktide arvu ette antud testprogrammi rikete katte viimiseks 100%-ni ja
- hinnata millise minimaalse testpunktide arvu juures 100%-ne rikete kate on saavutatav.

Käesoleva magistritöö tulemuste põhjal on kirjutatud artikkel [2], mis on esitatud k.a. sügisel toimuvale konverentsile Baltic Electronics Conference, 2014.

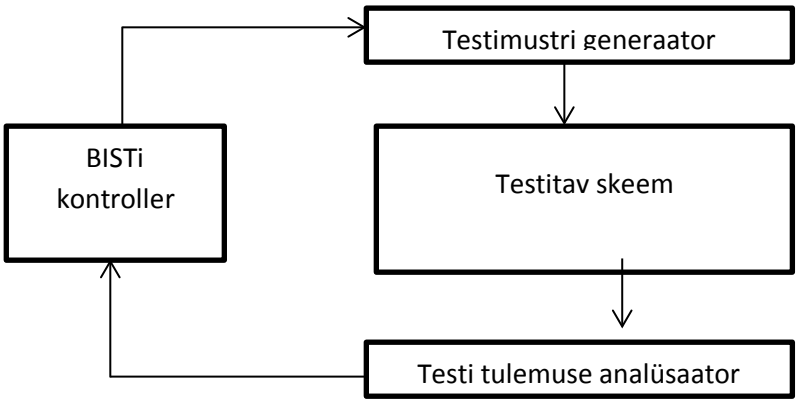
2. Digitaalskeemide isetestimine ja skeemide testitavus

Alljärgnevalt on esitatud ülevaade digitaalsüsteemide ja -skeemide isetestimise meetoditest ning isetestimise kvaliteedi tõstmise võimalustest.

2.1. Isetestimine juhuslike arvude kasutamisega

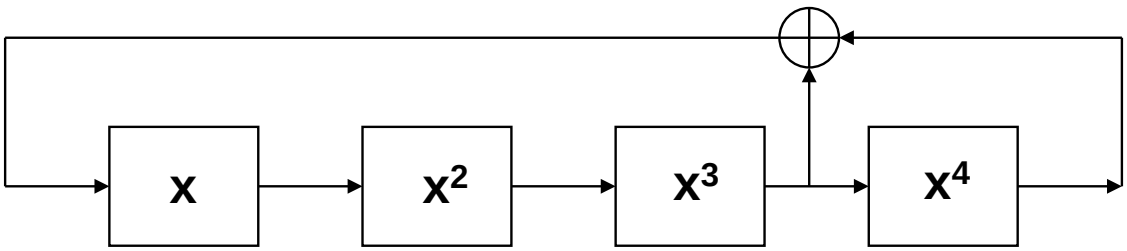
Traditsioonilistes BIST arhitektuurides kasutatakse testide genereerimiseks tüüpiliselt kas lineaarse tagasisidega nihkeregistrit (LFSR – Linear Feedback Shift Register) või multifunktsionaalseid registreid nagu BILBO (Built-in Logic Block Observer) [1]. Põhiideeks on siin kiibil asetsevat riistvara kasutades genereerida pseudojuhuslikke testvektoreid testitava skeemi (CUT – Circuit Under Test) jaoks ning seejärel testitava skeemi väljundeid analüüsida.

Skeemide isetestimine (edaspidi nimetame seda BISTiks) on nii elektronskeemidesse, süsteemidesse, kui ka kiipidesse sisse ehitatud seade, mis aitab testitaval skeemi osal (näiteks ALU) iseseisvalt, ilma kõrvalise abita, ennast ise testida ja vigu tuvastada. Klassikaline, kuid ka senini kasutav BIST koosneb kindlasti neljast üksusest: BISTi kontroller, pseudojuhuslike testvektorite generaator, testitav skeem ise ja testide tulemuste analüsaator (sageli kasutatav nn. signatuuranalüsaator). Abstraktselt ülesjoonistatud , kuid reaalsete osade ja suhtlussuundadega skeem on järgmine (Joon. 1):



Joonis 1. Tüüpiline isetestiv digitaalskeem sisse ehitatud testigeneraatori ja testitulemuste analüsaatoriga

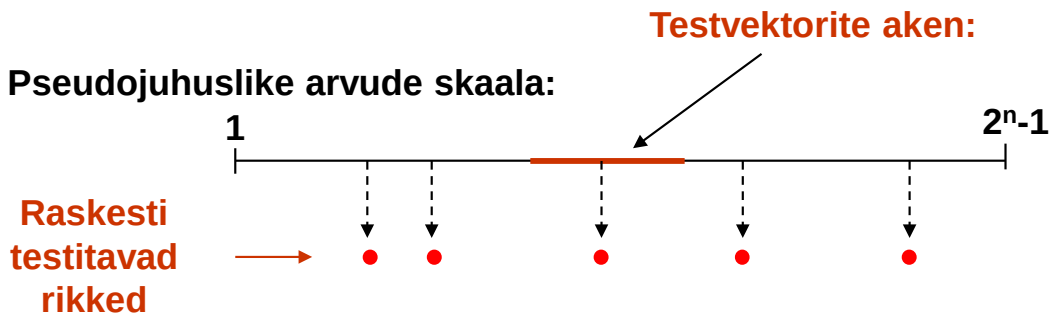
Pseudojuhuslike testide generaatorid ehitatakse harilikult lineaarse tagasisidega nihkeregistrite baasil (LFSR – Linear Feedback Shift Register), kus lineaarse tagasiside eesmärgiks on heade omadustega (võimalikult ühtlase jaotusega) 0-ja 1-signaalide genereerimine. Jooonisel 2 on näidatud 4 triggeriga lineaarse tagasisidega nihkeregister. Iga registri olek väljastab ühte konkreetset pseudojuhuslikku testvektorit. Iga järjekordne registri sisu nihe tähendab uut väljastatavat pseudojuhuslikku testvektorit. Kui nihkeregistri pikkuseks on *n*, siis antud seade on võimeline genereerima (sõltuvalt tagasisidede konfiguratsioonist) maksimaalselt 2^n-1 juhuslikku pseudojuhuslikku vektorit Vektorit, kus kõikides järkudes on 0, pole võimalik kasutada, sest sellest olekust poleks nihkeoperatsiooni tõttu võimalik välja pääseda. Harilikult on kasutatav testvektorite arv *N* palju väiksem, kui see oleks võimalik: $N << 2^n$-1.



Joonis 2. Lineaarse tagasisidega nihkeregister

„*Pseudojuhuslik*“ (aga mitte „*juhuslik*“) testvektorite jada tähendab siin seda, et genereeritavad signaalid 0 ja 1 on küll ühtlaselt jaotatud, aga siiski nende seaduspära on generaatori konkreetsest skeemist tulenevalt tuvastatav. See aga tähendab, et pseudojuhuslik jada on alati korduv, ehk siis üks ja see sama, mis on eelduseks sellele, et ka õigesti töötava testitava skeemi oodatav reaktsioon (etaloon) on alati üks ja see sama. Täiesti juhuslike signaalide kasutamisel poleks oodatavate (etaloonsignaalide) puudumise tõttu testimine võimalik.

Pseudojuhuslike arvude generaatori puuduseks on see, et ta ei sobi nn. „raskesti testitavate rikete“ avastamiseks. Olukorda ilmestab Joonis 3, kus vaadeldakse kogu võimalike pseudojuhuslike arvude vahemikku, millest kasutatakse ära vaid üsna väikest osa (vt. „Testvektorite aken“). Joonisel on näidatud 5 raskesti avastatavat riket, millistest igat ühte on võimeline avastama vaid üksainus testvektor antud skaalal. 5-st rikkest vaid ühte katab valitud testvektorite aken, mistõttu kõik ülejäänud 4 jäävadki avastamata. Ka siis kui sama pikkusega akent nihutada skaalal, ei paraneks olukord. Rikete täielik kate oleks võimalik vaid akna ulatuse ehk siis pseudojuhusliku testi pikkuse suurendamine, mis aga tähendaks ka testimise aja pikenemist.

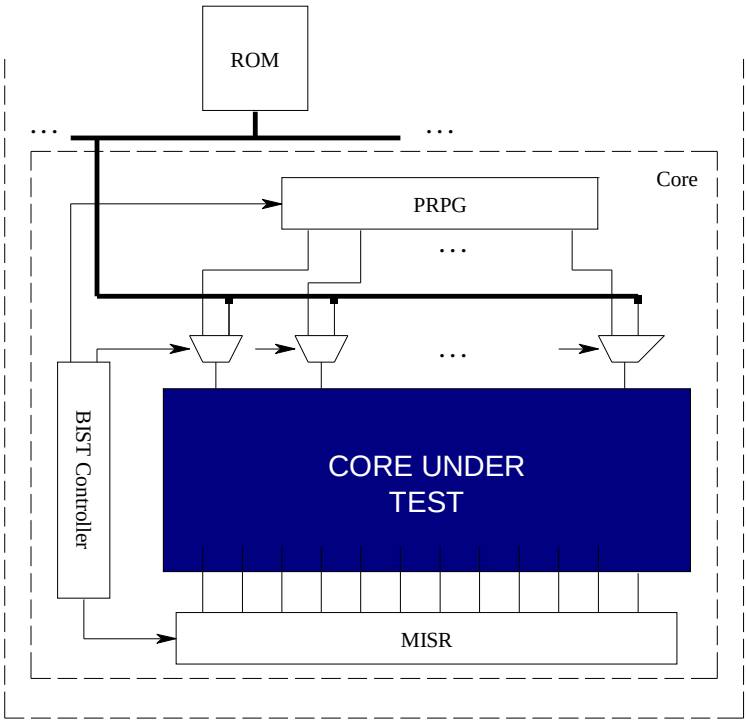


Joonis 3. Testimine pseudojuhuslike arvudega

Olukorrast väljapääsuks oleks antud pseudojuhusliku testi (ehk siis antud akna) täiendamine veel 4 deterministliku testvektoriga, mis oleksid võimelised näidatud „halbu“ rikkeid avastama. Niisugust kahte tüüpi testide ühitamist nimetatakse hübriidseks testimiseks.

2.2. Hübriidsed isetestimise meetodid

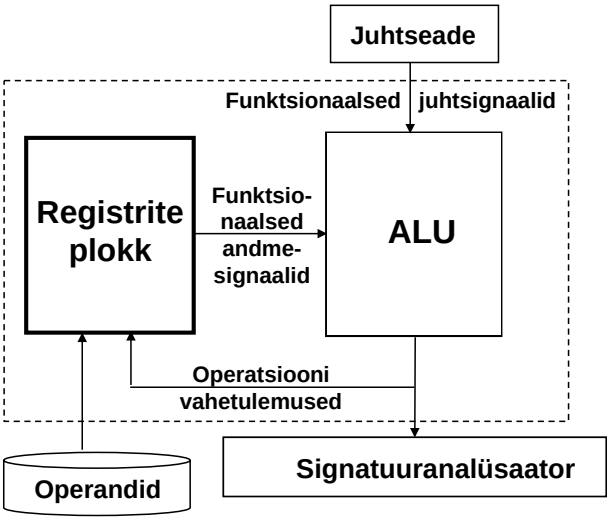
Hübriidse isetestimise näide, kus pseudojuhuslik testimine on ühitatud deterministliku testimisega, on esitatud Joonisel 4. Skeemi „Core Under test“ sisendites on 2-kanaliga multiplekserid, kus ühe kanali kaudu saadetakse signaale pseudojuhuslike arvude generaatorist PRPG (Pseudorandom Pattern Generator) ja teise kanali kaudu deterministlikke testsignaale, mis on salvestatud nt püsimälusse ROM. Multiplekserite ümberlülitamist juhib juhtseade BIST Controller. Testimise reaktsioonid (testitava skeemi väljundsignaalid) võtab aga vastu multisisendiline signatuuranalüsaator MISR (Multiple Input Signature Analyzer).



Joonis 4. Hübriidne isetestimine [10]

Üritades üle saada traditsioonilise BIST-i probleemidest, nagu lisa-aparatuuri vajadus LFSR-generaatori näol, on isetestimise alternatiiviks saanud funktsionaalse isetestimise idee. Termin „funktsionaalne BIST“ (FBIST) kirjeldab meetodit juhtimaks funktsionaalseid mooduleid sellisel viisil, et nad genereeriksid töö käigus testvektoreid, millega saab testida struktuurseid vigu teistes süsteemi osades. Tegemist on paljulubava ideega testimaks keerukaid digitaalskeeme väiksema hulga lisaloogika ning jõudluse vähendamiseta.

Käesolevas töös ongi loobutud pseudojuhuslike testvektorite generaatori kasutamisest, seades eesmärgiks aparatuurse kokkuhoiu ühelt poolt ja reaalsete töösignaalide kasutamisest tuleneva testimise kvaliteedi tõusu teiselt poolt. Alternatiivseks lahenduseks oleks siin funktsionaalne isetestimine, mida illustreerib Joonis 5.

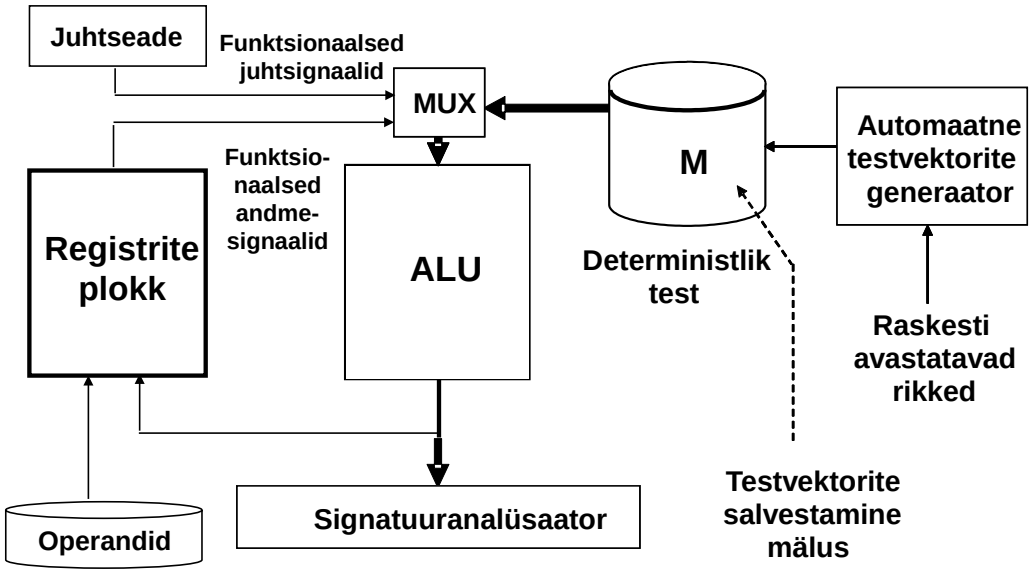


Joonis 5. Funktsionaalne isetestimine[10]

Joonisel 5 on esitatud operatsioonseade, mis koosneb juhtautomaadist ja operatsioonautomaadist. Omakorda viimane koosneb registrite plokist ja kombinatoorsest aritmeetika-loogikaplokist ALU. Isetestitavaks objektiks on siin ALU. Funktsionaalsed testvektorid kujunevad siin suvalise operatsiooni ehk mikroprogrammi (nt korrutamise) käigus registrite plokist tulevate andmesignaalide ja juhtseadmest tulevate juhtsignaalide kogumina (ehk siis niisuguste kogumite jadana). Ainsaks lisaseadmeks on selles skeemis ALU väljundeid monitooriv signatuuranalüsaator. Kõiki testsignaale võib antud juhul vaadelda samuti pseudojuhuslike signaalidena nagu eelpool vaadeldud LFSR-genraatori puhul, ainult selle vahega, et praegu tekivad testid puhtal kujul töö käigus ja LFSR-i ei ole vaja.

Kahjuks ka antud juhul ei pruugi funktsionaalne test olla piisava kvaliteediga. Olukorrast väljapääsuna võib ka siin kasutada hübriidset lähenemisviisi, kus funktsionaalsete töösignaalide kõrval kasutatakse täiendavaid deterministlikke testvektoreid, milliste eesmärgiks oleks „raskesti avastatavate rikete“ kinni püüdmine.

Joonisel 6 on näidatud, kuidas on võimalik Joonisel 5 näidatud skeemist kujundada ümber hübriid-funktsionaalne isetestiv seade.

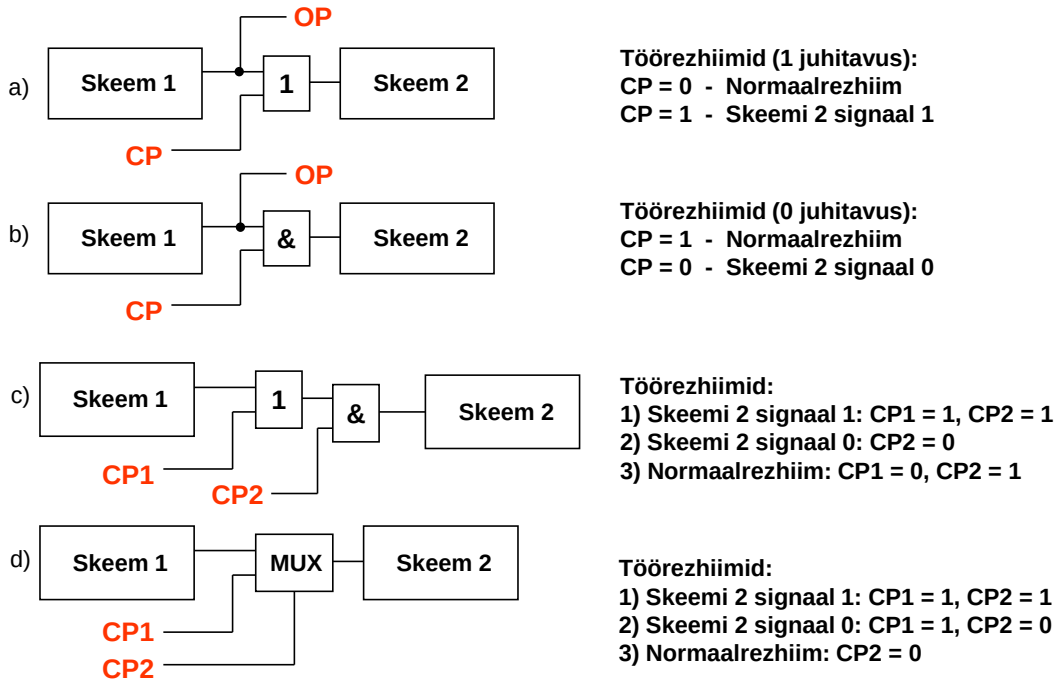


Joonis 6. Hübriid-funktsionaalne isetestimine[10]

ALU sisendisse on paigutatud multiplekserite plokk analoogiliselt skeemile Joonisel 4, mille kaudu saabuvad ALU sisendisse nii töörezhiimi funktsionaalsed vektorid kui ka mälust loetavad deterministlikud testvektorid. Vastavalt sellele koosneb kogu test kahest osast: funktsionaalsest testseansist ja deterministlikust testseansist.

2.3. Digitaalskeemide testitavus ja selle parandamine

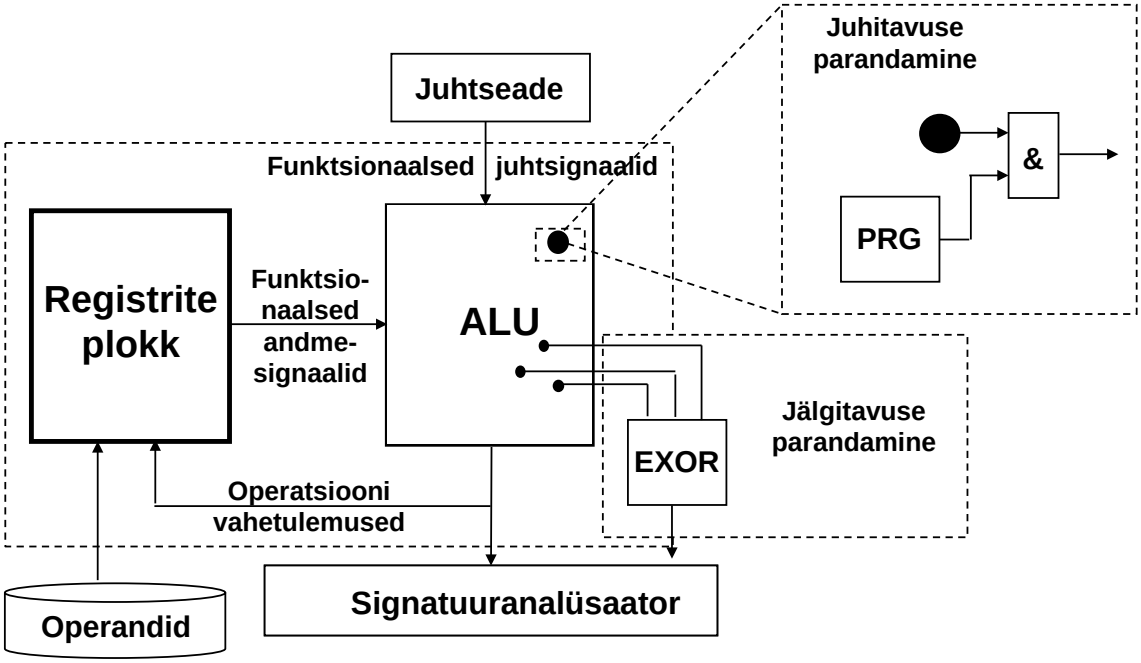
Rikete katte parandamiseks nii traditsioonilise BISTi kui ka funktsionaalse BISTi puhul on hübriidtestimise põhimõtte kõrval võimalik parandada skeemi testitavust ka lisatestpunktide sisseviimise näol, kas siis testitava skeemi sisepunktide jälgitavuse või juhitavuse parandamiseks [4]. Vastavalt sellele, on ka testpunkte kahte sorti – juhtimispunktid ja jälgimispunktid.



Joonis 7. Testitavuse parandamise meetodid[10]

Kui skeemi sisepunktid on „sügaval“ skeemi sees, siis võib skeemi käitumise jälgimise lihtsustamiseks vastavatesse kohtadesse lisada väljaviike. Mitte alati aga pole võimalik skeemi testitavust parandada ainult väljaviikude abil. Nendel juhtudel, kui rikke avastamiseks vajalikku bitikombinatsiooni ei teki ühegi testprogrammis sisalduva testvektori korral, tuleb parandada skeemi juhitavust, et oleks võimalik vajalikke bitikombinatsioone saavutada täiendava välise juhtimise abil. Niisuguseks väliseks juhtimiseks võib kasutada nii konkreetseid signaale 0 või 1, kui ka pseudojuhuslikke signaale analoogiliselt pseudojuhusliku testimisele. Erinevuseks on ainult see, et juhitavuse parandamiseks vajalike lisasignaalide arv on palju väiksem kui pseudojuhusliku testimise korral.

Näited skeemide jälgitavuse parandamiseks väljaviikude OP abil on toodud Joonistel 7a ja 7b. Joonisel 7a demonstreeritakse juhitavuse parandamist signaali 1 saamiseks testpunktis VÕI-elemendi abil, Joonisel 7b parandatakse juhitavust signaali 0 saamiseks JA-elemendi abil ning Joonisel 7c on näha, kuidas on võimalik skeemi juhitavust parandada nii signaalide 0 kui ka 1 saamiseks, kasutades mõlemat loogikaelementi. Joonisel 7d on näidatud, kuidas saab juhitavust parandada multiplekseri abil.



Joonis 8. Testitavuse parandamine funktsionaalse isetestimise puhul [10]

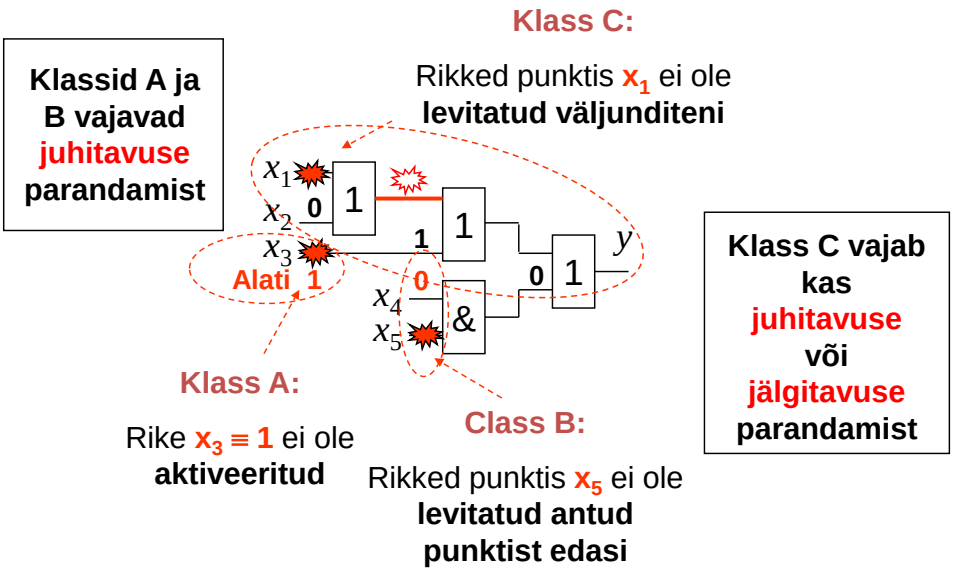
Joonisel 8 illustreeritakse, kuidas on võimalik tõsta isetestitava skeemi testimise kvaliteeti testpunktide lisamise abil nii juhitavuse kui jälgitavuse parandamiseks. Juhitavuse parandamiseks kasutatakse juhtimissignaalide allikana pseudojuhuslike signaalide generaatorit PRG (Pseudorandom Generator). Jälgitavuse parandamisel aga selleks, et vältida lisaväljaviikude vajadust kiibist endast ühendatakse jälgitavad punktid otse või EXOR elementide abil signatuuranalüsaatoriga.

2.4. Digitaalskeemide testitavus ja selle parandamine

Rikete simuleerimise abil määratakse kindlaks, millised rikked on skeemis antud testprogrammi või antud testvektorite hulga poolt avastataud. Avastamata rikked on otstarbekas klassifitseerida kolme järgnevasse klassi:

- Klass A: rikked, mida ei aktiveeri ükski testvektor antud testvektorite hulgast
- Klass B: rikked, mida võidakse küll aktiveerida, aga mis ühegi aktiveeriva testvektori puhul ei levi oma asukohast kaugemale
- Klass C: rikked, milliseid võidakse küll aktiveerida, ja mis võivad ka oma asukohast kaugemale levida, aga mis ühegi testvektori puhul ei levi ühegi skeemi väljundini.

Klassi A ja B kuuluvaid rikkeid on võimalik antud testvektorite hulga puhul avastatavaks teha üksnes täiendavate testpunktide sisse viimise abil, mis parandavad juhitavust. Seevastu klassi C kuuluvaid rikkeid on võimalik testitavaks teha nii juhitavuse kui ka jälgitavuse parandamise abil. Joonis 9 illustreerib rikete kuuluvust klassidesse A, B ja C.



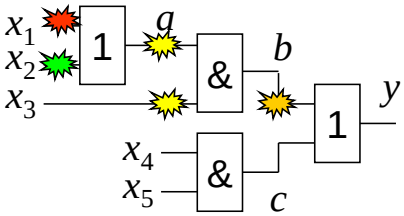
Joonis 9. Rikete klassid A, B ja C [10]

Joonisel 10 on näitena toodud kolmest testvektorist koosnev test ja ära märgitud 5 riket, mida antud test ei avasta. Rikked on ära näidatud ka skeemil ja klassifitseeritud klassidesse A, B ja C. Kõrval toodud tabelis on ära märgitud testvektorite poolt tekitatud signaalide väärtused skeemis ja rikete tabel, mis näitab, millised rikked on avastatud (1 – tähendab konstantriket 1 ja 0 tähendab konstantriket 0) ning millised rikked on jäänud avastamata.

Antud test:

No	Testvektorid								Rikete tabel							
	Sisendid					Sise-punktid			Sisendid					Sise-punktid		
	1	2	3	4	5	a	b	c	1	2	3	4	5	a	b	c
1	0	0	1	0	1	0	0	0	1	1	-	1	-	1	1	1
2	0	1	0	1	1	1	0	1	-	-	-	0	0	-	-	0
3	0	1	0	1	0	1	0	0	-	-	1	-	1	-	1	1

$x_1/0$ $x_2/0$ $x_3/0$ $a/0$ $b/0$



Avastamata rikked:

Klass	Rikked	Puuduvad signaalid	
A	$x_1/0$:	$x_1 = 1$	puudub
A	$b/0$:	$b = 1$	puudub
B	$x_3/0$:	x_3 a = 11	puudub
B	$a/0$:	x_3 a = 11	puudub
C	$x_2/0$:	x_1x_2= 01	OK

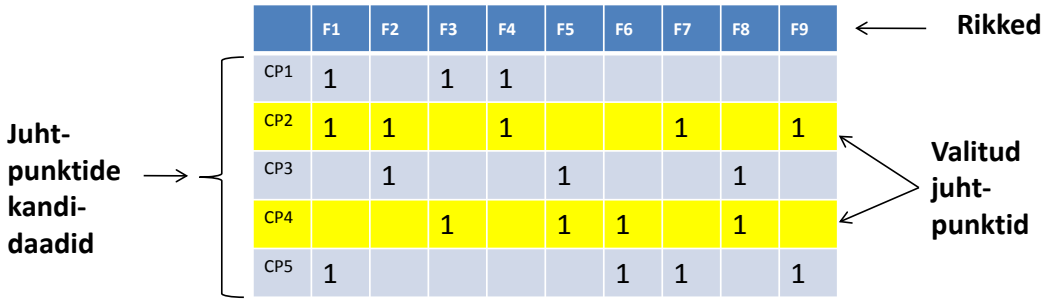
Joonis 10. Kolm testvektorit ja nende poolt avastamata jäänud 5 riket [10]

Joonisel 10 ära toodud skeemis on ära märgitud kohad, millistes on mingid rikked jäänud avastamata. Need kohad on ka kandidaatideks, kuhu oleks võimalik testpunkte organiseeridsa testitavuse parandamiseks. Samas ei ole need punktid kõik vajalikud, sest ühe testpunkti sisse viimine võib üheaegselt võimaldada mitme rikke avastamist.

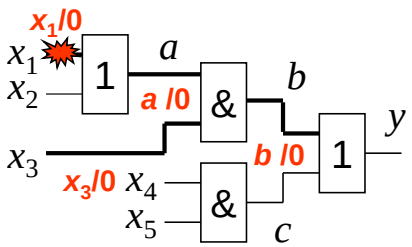
Testpunktide arvu minimeerimiseks tulebki valida kõige sobivamad testpunktid, mis kõige paremini aitaksid kõiki rikkeid kokku avastada. Testpunktide arvu minimeerimiseks tuleb ehitada testpunktide tabel, kus on näidatud, milliseid rikkeid võimaldab iga testpunkt avastada. Tabeli minimaalne kate aitakski määrata minimaalse testpunktide hulga. Sellise tabeli ehitamiseks tuleb igasse punkti, kus mingi rike on jäänud avastamata sisestada testpunkt, simuleerida uuesti rikkeid ja leida, milliseid rikkeid antud testpunkt võimaldab avastada. Niisugused testpunktide tabelid tuleks ehitada nii juhtpunktide määramiseks rikete klasside A ja B likvideerimiseks ning jälgimispunktide määramiseks rikete klassi C likvideerimiseks, juhul kui see juba pole likvideeritud vajalike juhtpunktide sisse viimise abil.

Joonisel 11 on esitatud näitena testpunktide tabel skeemi testitavuse parandamiseks testpunktide skeemi viimise abil. On näidatud ka minimaalse katte leidmine tabelis,

mille tulemusena on välja valitud kaks testpunkti CP2 ja CP4. Niisugused tabelid tuleb ehitada nii juhtimise parandamiseks rikete klasside A ja B likvideerimise eesmärgil, kui ka jälgitavuse parandamiseks (juhul kui koos klasside A ja B likvideerimisega pole ära kadunud ka klass C).



Joonis 11. Testpunktide tabel juhitavuse parandamiseks



Avastamata rikked:

Klass A: $x_1/0$, $b/0$

Klass B: $x_3/0$, $a/0$,

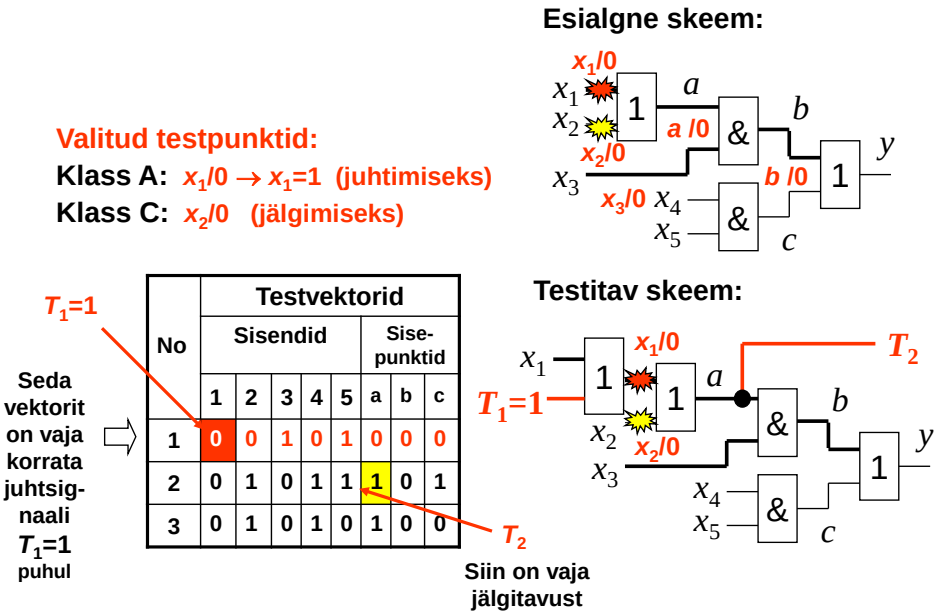
Testpunktide tabel:

Valitav punkt	Avastamata rikked			
	$x_1/0$	$x_3/0$	$a/0$	$b/0$
Potent-siaalsed test-punktid: $x_1=1$	+	+	+	+
$x_3=1$		+	+	+
$a=1$		+	+	+
$b=1$				+

No	Testvektorid							
	Sisendid					Sise-punktid		
	1	2	3	4	5	a	b	c
1	0	0	1	0	1	0	0	0
2	0	1	0	1	1	1	0	1
3	0	1	0	1	0	1	0	0

Joonis 12. Testpunkti valik 4 kandidaadi valik (ülal toodud näites) [10]

Ülal toodud näite jaoks, kus 5 riket jäi avastamata, on joonisel 12 toodud testpunktide tabel juhtimise parandamiseks, mille jaoks on leitud minimaalne kate üheainsa testpunkti x_1 näol, mille sisse viimine aitab testitavaks teha kõik 4 klassidesse A ja B kuuluvad rikked.



Joonis 13. Juhtpunkti ja jälgimispunkti skeemi sisseviimine [10]

Joonis 13 ilustreerib ülal toodud näite puhul lõplikku testitavuse parandamise lahendust, mis seisneb ühe juhtimispunkti ja ühe jälgimispunkti skeemi sisse viimises.

3. Meetodi ja algoritmi väljatöötamine digitaalskeemide juhitavuse parandamiseks testpunktide sisseviimise abil

Käesolevas peatükis on välja töötatud meetod ja algoritm digitaalskeemide testitavuse parndamiseks. Vaadeldakse ainult juhitavuse parandamist. Juhitavuse valimisel uurimisobjektiks tõusid esile kaks motiivi:

- juhitavuse parandamine võimaldab saavutada testitavust 100%-lise rikke katte näol alati, mida aga ei võimalda alati jälgitavuse parandamine:
- kontsentreerumine üksnes juhitavuse aspektile võimaldab paremini seada fookust, teiste sõnadega uurida üksnes ühte testitavuse tõstmise komponenti (jälgitavus oleks seejärel samuti iseseisev teine uurimisobjekt);
- fookuse täpsem seadmine võimaldab paremini selgitada, kuidas üks või teine testitavuse mõjutamise meetod toimib.

3.1.Eksperimentaalkeskkonna kujundamine

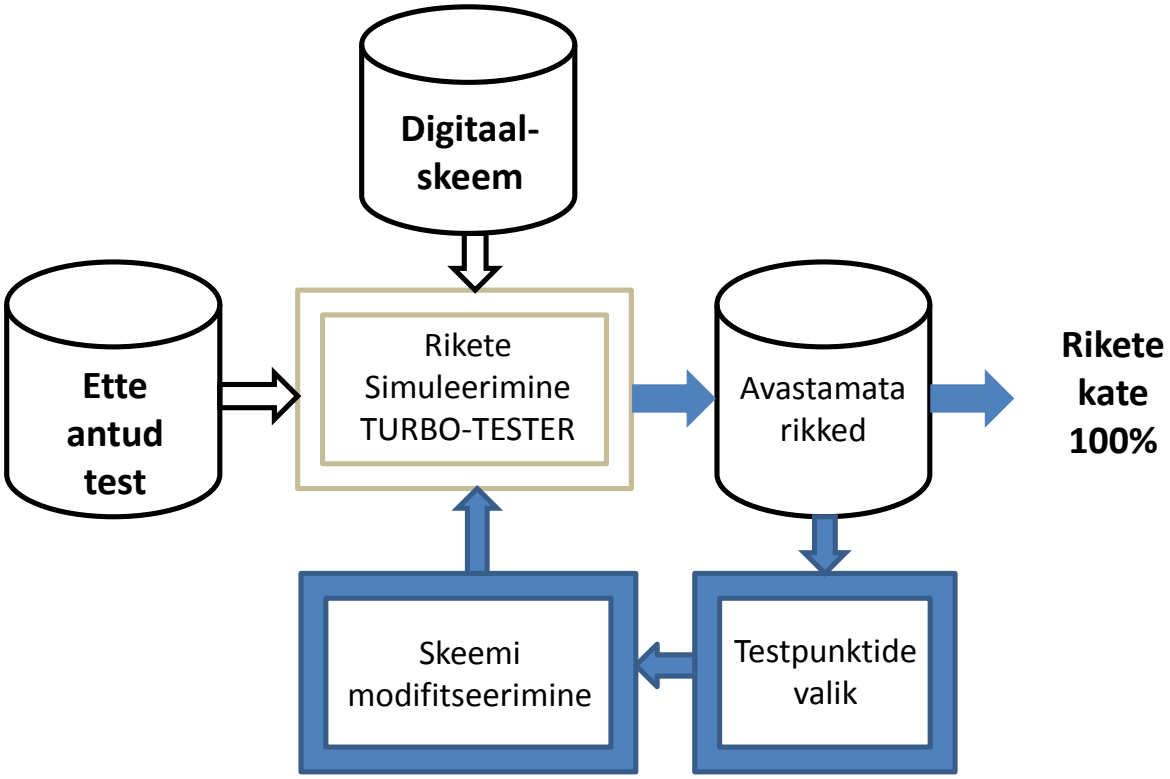
Skeemi testitavuse analüüsiks ja testpunktide sisestamise efekti kindlaks tegemiseks loodi tarkvarakeskkond, mis koosnes järgmistest tööriistadest (vt. Joonis 9):

- rikete simulaator (Turbo-Testri tarkvara),
- programm testpunktide valikuks ja
- skeemi modifitseerimise programm sisseviidud testpunktide efekti kindlaks tegemiseks.

Kaks viimast programmi töötati välja antud magistritöö raames.

Töö eesmärgiks oli parandada etteantud testprogrammi (testvektorite jada) jaoks skeemi juhitavust täiendavate testpunktide sisseviimise abil testide kvaliteeti nii, et rikete kate jõuaks 100%-ni. Rikete mudelina kasutati klassikalist konstantsete rikete

mudelit. Rikete simuleerimiseks ja rikete katte arvutamiseks kasutati Arvutitehnika instituudis välja töötatud tarkvara Turbo-Tester tarkvaratööriistade kasti kuuluvat spetsiaalset rikete simulaatorit [5-8].



Joonis 14. Tarkvarakeskkond skeemi testitavuse analüüsiks ja parandamiseks

Vastavalt Joonisel 14 esitatud keskkonnale tuleb kõigepealt kindlaks teha ette antud funktsionaalse testi rikete kate ja määrata kindlaks avastamata rikked.

3.1.1.Testpunktide valik

Üheks magistritöö tulemuseks oli testpunktide valiku algoritmi ja programmi koostamine.

Tähistame skeemis oleva rikete hulga sümboliga *R* ja testi kuuluvate testvektorite hulga sümboliga *T*. Rikete simuleerimise tulemusena ehitatakse rikete tabel maatriksina RT = $||e_{ij}||$, kus

$i = 1,2, ..., n$; n – on rikete arv,

j = 1,2, ..., m; m – on testvektorite arv ja

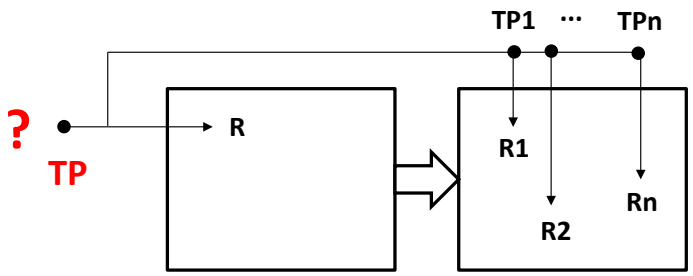
e_{ij} = 1, kui testvektor $t_i \in T$ avastab riket $r_j \in R$ ja e_{ij} = 0, kui ei avasta.

Tähistame sümboliga k_j korduste arvu, millega avastatakse riket r_j. Kui k_j = 0, siis antud testvektrorite hulk T ei avasta riket r_j kordagi.

Olgu $R(k) \subseteq R$ – rikete hulk, milliseid antud test avastab k või vähem kordi. Kui k k = 0, siis $R(k)$ sisaldab üksnes avastamata jäänud rikkeid.

Traditsioonilise testpunktide määramise meetodi [4] puhul valitaksegi testpunktid rikete hulga $R(k)$ järgi, kus k = 0. Hulga $R(k)$, k = 0, ja rikete tabeli RT abil ehitatakse testpunktide tabel ja seejärel leitakse selle minimaalne kate minimaalse testpunktide hulga välja valimise teel, nii nagu on seda kirjeldatud punktis 2.4.

Käesolevas töös on töötatud välja meetod, mis võimaldab veelgi rohkem testpunkte minimeerida võrreldes meetodiga [4]. Siin lähtutakse avastusest, mis töös läbi viidud eksperimentide puhul selgus, et juhitavuse parandamine mitte üksnes punktides, kus rikked on jäänud avastamata, vaid ka punktides, kus rikkeid on avastatud väga vähe arv kordi, võimaldab üldkokkuvõttes veelgi vähendada vajalikku testpunktide arvu.



Joonis 15. Potentsiaalsete testpunktide üheaegne mõju paljude rikete avastamisele

Joonisel 15 on illustreeritud meetodi kasutamise motivatsiooni. Vaatleme olukorda, kus rike R on testitud ja seetõttu testpunkti TP, mille kaudu riket aktiveeritakse pole justkui vaja. Sama signaal aktiveerib aga üheaegselt ka rikkeid R1, R2, ..., Rn. Kui

ette antud funktsionaalne test avastab riket R vaid korra üheainsa testvektori abil (või väga vähe arv kordi), siis võib osa rikkeid R1, R2, ..., Rn jääda testimata, sest see ainus testvektor ei pruugi kõiki neid rikkeid üheaegselt aktiveerida. Siit aga tuleneb juba vajadus viia skeemi testpunktid TP1, TP2, ..., TPn.

Uus idee seisneb selles, et selle asemel, et lisada testpunktide kandidaatide hulka TP1, TP2, ..., TPn, võiks hoopis lisada sinna üheainsa testpunkti TP, mis mõjutab üheaegselt kõiki avastamata jäänud rikkeid, sellele vaatamata, et rikke R enda jaoks seda testpunkti pole vaja (R on juba testitud).

Nimetatud avastus juhtis võimalusele ehitada testpunktide tabelit rikete hulkade $R(k)$ põhjal, kus $k \geq 0$. Eksperimentide tulemused kinnitasid hüpoteesi, mis ennustas, et parameetri k suurendamise korral väheneb (või jääb endiseks) vajalik testpunktide arv.

Võimalus parameetri k suuruse muutmiseks ja testpunktide valimiseks erinevate k väärtuste puhul viidi ka testpunktide valiku programmi.

3.1.2. Testpunktide emuleerimine skeemi modifitseerimise asemel

Teiseks magistritöös välja töötatud tulemuseks on originaalne testpunktide emuleerimise meetod skeemi modifitseerimise asemel testpunktide toime katsetamiseks. Idee seisneb järgnevas: selle asemel, et skeemi reaalselt modifitseerida testpunktide sisseviimise teel, modifitseeritakse hoopis simuleerimistabelit, milline tegevus on palju lihtsam ja kiirem.

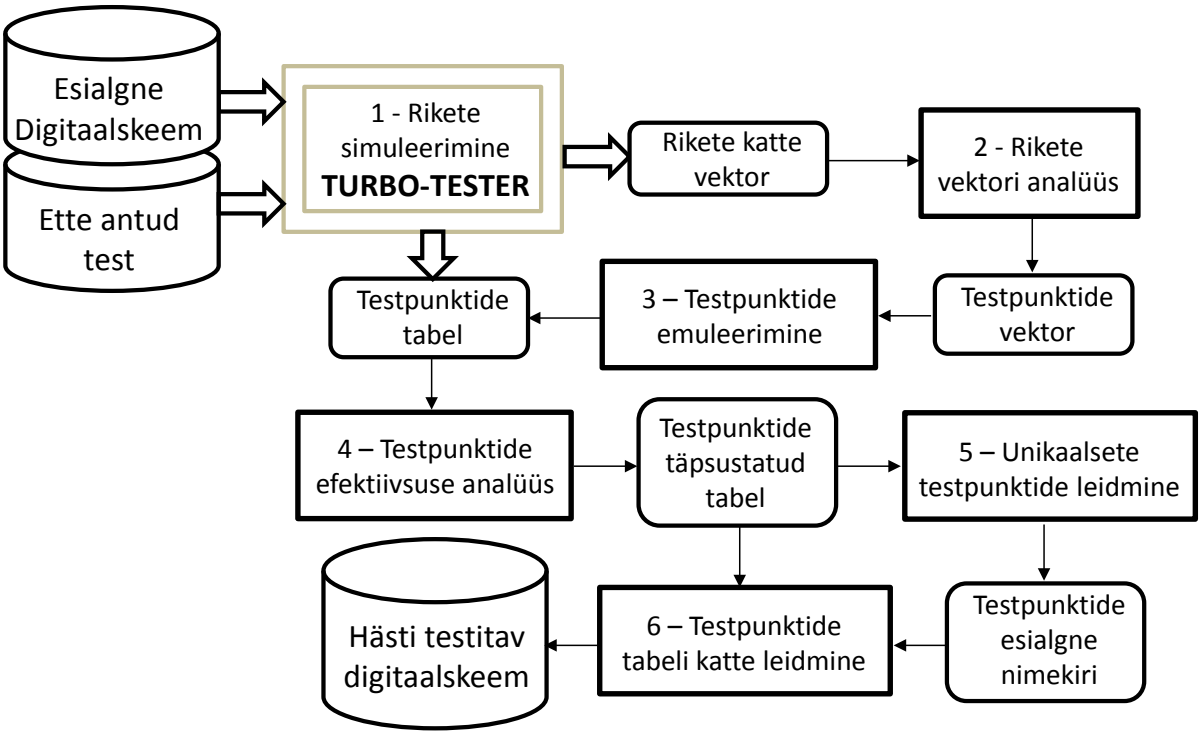
Olgu antud simuleerimistabel $ST = ||s_{ij}||$, mille ridadeks i on testvektorid $t_i \in T$, veergudeks j skeemipunktid s_j ning muutuja s_{ij} tähistab signaali väärtust testvektori t_i ajal skeemipunktis s_j. Kuna signaalide s_{ij} arvutamine testvektorite simuleerimisel toimub indeksi j kasvamise järjekorras (skeemipunktid on vastavalt järjestatud), siis on võimalik juhtimispunkti toimet skeemipunktis s_j imiteerida kogu veeru j vektori asendamisega kas konstantide 0 või 1-ga või pseudojuhuslike signaaliväärtustega, vastavalt juhtimispunkti rollile skeemi juhitavuse parandamisel. Millisele testvektorile konkreetselt juhtimissignaal mõju avaldab selgub kõigi testvektorite simuleerimise käigus.

Kirjeldatud testpunktide toime imiteerimise võimalus on viidud kirjeldatava keskkonna skeemi modifitseerimise programmi. Programm modifitseerib etteantud testpunkti jaoks esialgset simuleerimistabelit *ST* ja arvutab selle ümber saades uue modifitseeritud simuleerimistabeli *ST'*. Viimase jaoks viiakse läbi rikete simuleerimine ja tehakse kindlaks, milliseid rikkeid antud testpunkti sisseviimine võimaldab avastada.

Kui kõikide rikete jaoks hulgast $R(k) \subseteq R$ on testpunktide imiteerimine läbi viidud, siis ehitatakse testpunktide tabel ning viiakse läbi tabeli katte minimeerimine. Saadav tulemus annab vajaliku testpunktide hulga, mis võimaldab ette antud testidega saavutada 100%-line rikete kate.

3.2. Algoritm ja programm skeemide juhitavuse parandamiseks

Magistritöö tulemusena väljatöötatud detailne tarkvaratööriistade süsteem on esitatud joonisel 16. Süsteemi kerkseke tööriistaks on Arvutitehnika instituudis välja töötatud Turbo-Testri rikete simulaator, millele on juurde lisatud viis uut programmi: 2- rikete vektori analüüs, 3 – testpunktide emuleerimine, 4- testpunktide efektiivsuse analüüs, 5 – unikaalsete testpunktide leidmine ja 6 – testpunktide tabeli katte leidmine.



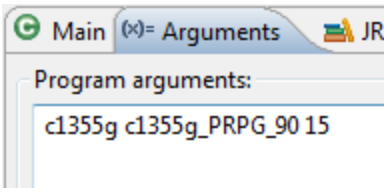
Joonis 16. Magistritöös välja töötatud tarkvaratööriistade süsteem (programmid 2-5)

Süsteemi sisendiks on esialgne ette antud digitaalskeem, mille juhitavust on vaja parandada, ning ette antud funktsionaalne test, mille rikete kate on väiksem kui 100%. Süsteemi väljundiks on uus hästitestitav digitaalskeem, mille puhul ette antud funktsionaalne test garanteerib 100%-lise rikete katte. Konkreetset uut skeemi küll ei esitata, kuid antakse vajalike testpunktide loetelu, mis on vajalik skeemi sisse viia 100%-lise rikete katte saamiseks.

Järgnevalt on esitatud täpsemalt süsteemi kuuluvate tööriistade (eelpool nimetatud 5 programmi) kirjeldused, mis seisnevad programmide sisendite ja väljundite defineerimises ning programmide poolt täidetavate ülesannete kirjeldamises.

3.2.1. Rikete simuleerimine.

a) Programmi sisendiks on etteantud digitaalskeemi kirjeldus (.agm fail) ja etteantud funktsionaalse testi ehk testvektorite massiivi kirjeldus (.tst fail). Vastavad failid tuleb programmile käivitamisel käsitsi ette anda (testimise alustamiseks tuleb kirjutada sisend tekst kindlas formaadis : [Skeemi faili nimi] [Rikete faili nimi] [Limit]). Joonisel 17 on näha programmi käivitust koodi arendamisel Eclipse programmi terminali kasutades.



Joonis 17. Programmi koodi käivitamine programmi Eclipse abil.

b) Programmi väljundiks on rikete katte vektor „initTestedFaults“, milles on iga skeemipunkti kohta esitatud üks järgmistest sümbolitest: & - mõlemad konstantrikked on antud testiga avastatud, 0 – ainult konstant 0 rike on avastatud, 1 – ainult konstant 1 rike on avastatud ja X – ei kumbagi riket pole avastatud.

c) Funktsiooni käigus luuakse edasises töös kasutatav rikete katte vektor „initTestedFaults = new FaultVector(tstFile.readFaults())“ , kus tstFile on simuleeritud ette antud testide fail.

3.2.2. Rikete katte vektori analüüs

a) Funktsiooni sisendiks on eelneva, testide simuleerimise faasi väljund – vektor „initTestedFaults“. Vastav muutuja sisaldab tulemusi kõige esmasest testide simulatsioonil saadud failist (skeemi nimi.tst).

b) Funktsiooni väljundiks on ette antud testi poolt avastamata jäänud rikete vektor „nodsToTest“, mida saab interpreteerida potentsiaalse testpunktide vektorina. See on vektor skeemipunktide kohta, milliseid hakatakse edasises töös vajalike testpunktide kandidaatidena testitavuse parandamiseks kasutama.

c) Programmi ülesandeks on otsida testimata jäänud konstant ühe või konstant nulli rikkeid. Vastavalt eelpool punktis 3.1.1. toodud kirjelduasele, vastab see rikete hulgale R(k), kus k = 0. Programmi kasutamiseks lõpptulemuste optimeerimise eesmärgil, on viidud sisse parameeter limit, mis vastabki muutujale k, mida on võimalik programmi kasutamisel ette anda.

Programmi tööd iseloomustab kood:

```
for(int n=0; n<nodCount; n++){
```

```
                    if(saf[0][n] <= limit){
                          nodsToTest.add(n);
                          valToTest.add(true);
                    }
                    if (saf[1][n] <= limit){
                          nodsToTest.add(n);
                          valToTest.add(false);
                    }
                    if(saf[0][n] == 0 || saf[1][n] == 0)
nodsNotTested.add(n);
              }//end for n
```

Kasutatud tähised:

nodCount – kogu sõlmede arv skeemis.

Llimit – rikete avastamise kordumiste arv antud testi abil

3.2.3. Testpunktide emuleerimine

Programmis toimub testpunktide sisseviimine skeemi nende emuleerimise teel ja Turbo-Testeri simulaatori välja kutsumine ning rikete simuleerimine tingimustel, kus antud testpunktis on juhtimine (testitavus) vajalikul viisil parandatud.

a) Sisendiks on programmi 2 poolt genereeritud potentsiaalsete testpunktide hulk, mis esitatakse koodis vektoritega cpFaults ja cpLocal, testitavate sõlmede arvuga, testitava sõlme järjekorra numbri ehk asukoha nimetamisega ja avastamata jäänud rikke tüübiga (nodsToTest.get(i), valToTest.get(i))

b) Funktsiooni väljundiks on fail laiendiga .cp („control ponts“), kuhu on sissesalvestatud erinevate avastamata jäänud rikete (kui k = 0) või raskesti testitavate sõlmede (kui k > 0) andmed. Näiteks tähendab vastav kirje .cp failist, „5701 0“, et viietuhande seitsmesaja esimesele sõlmele tuleb anda juurdeviik sisendväärtusega 0, mis peaks suutma tuvastada „saf1“ , kuid alati ei saa kindel olla, et vastava juurdeviiguga soovitud viga saab ka koheselt tuvastatav.

c) Antud programmi abil toimub testpunktide emuleerimine etteantud testide simuleerimiseks. Emuleerimine tähendab seda, et skeemi ennast ei modifitseerita testpunkti sisse viimise abil, vaid modifitseeritakse simuleerimistabelit sel moel, kuidas on mõeldud testpunkti kasutada. Kui näiteks testpunktis nõutakse juhitavuse parandamist signaali 1 abil, siis modifitseeritaksegi simuleerimistabelit skeemi vastavas punktis selle signaali abil ja viiakse simuleerimine ise lõpuni justkui oleks sellesse punkti saadetudki signaal 1. Konkreetselt, uuritakse selle funktsiooni abil läbi kõik välja valitud skeemi sõlmed (for(int i=0; i<nodsToTest.size(); i++)) ning viiakse kokku (liidetakse) olemasolev vektor uue, testpunktiga genereeritud, vektoriga (cpLocal[i].or(cpFaults[i]);), kus

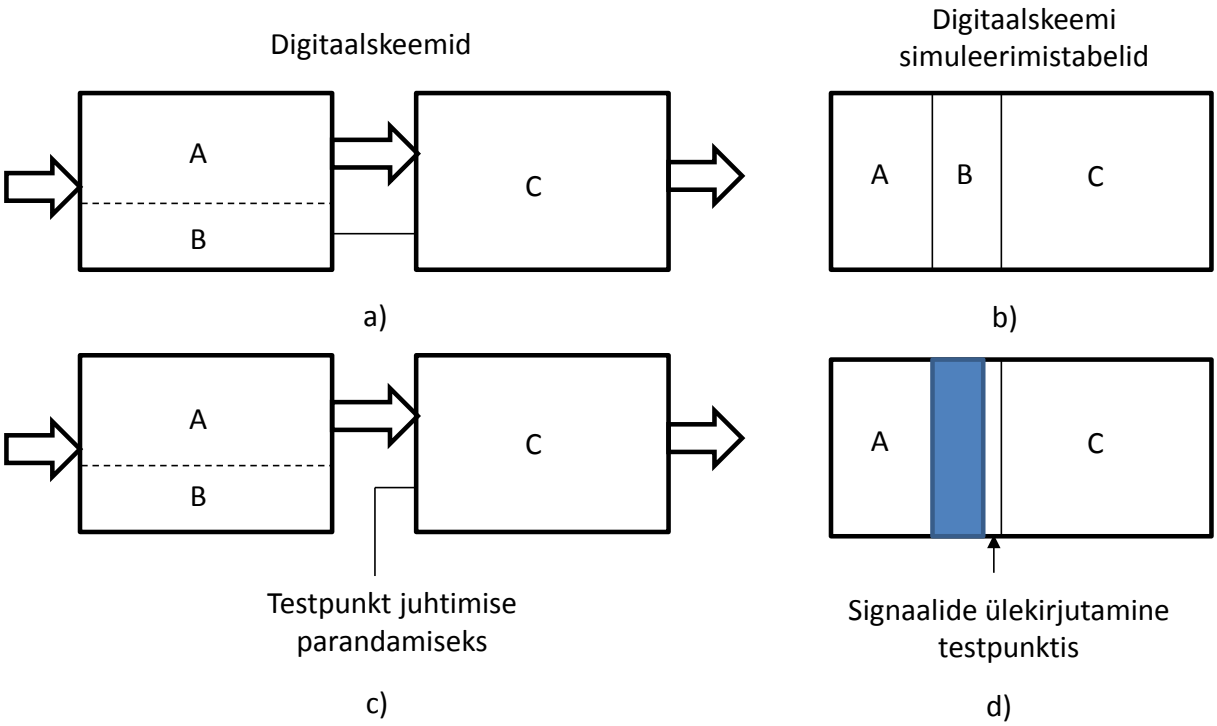
cpFaults[] = Testpunktide lisamine

cpLocal[] = Liidetud testpunktid + hetkel testitud väärtuste tulemused

Programmi tulemusena saame info selle kohta, kuidas toimuks rikete avastamine etteantud testprogrammiga iga testpunkti sisseviimise puhul eraldi. Infot esitab tabel, mille read vastavad testpunktidele ja veerud vastavad skeemi punktidele. Iga rida vastab antud juhul rikete katte vektorile, mis saadaks siis, kui antud testpunkt oleks skeemi sisendiks.

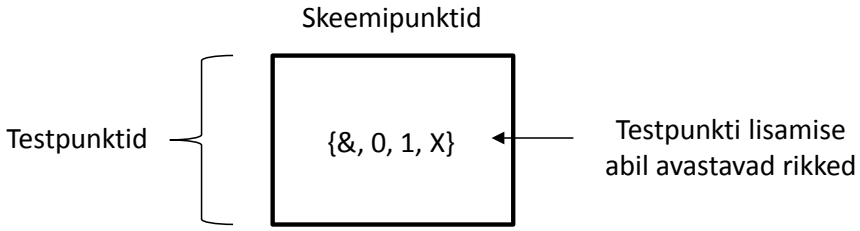
Testpunktide emuleerimise ideed kirjeldab Joonis 18, kus kirjeldatakse testpunkti sisse viimist kolmest plokist A, B ja C koosnevasse skeemi ploki C juhtimise parandamiseks. Esialgne skeem on esitatud Joonisel 18a ja skeemi simuleerimise tabelit ette antud testvektorite jaoks Joonisel 18b. Simuleerimise tabeli ridadele vastavad testvektorid ja veergudele skeemipunktid. Veerud on järjestatud selliselt, et igale veerule vastavaid signaale on võimalik arvutada vaid siis, kui eelnevatele veergudele vastavad signaalid on arvutatud. Skeem, millesse on viidud testpunkt, on esitatud Joonisel 18c, kus on näha, et plokk B väljund on katkestatud ja sellest väljuvaid signaale asendavad testpunkti kaudu skeemisaadetavad signaalid. Testpunktide emuleerimise ideed väljendab Joonis 18d, kus on näha, et ploki B signaalid (tabeli tume osa) on arvutamata jäänud ja ploki B väljundi signaale asendavad testpunkti signaalid.

Igale testpunktile sünteesitakse talle vastav simuleerimistabel, mille põhjal omakorda Turbo-testri rikete simulaator arvutab rikete tabeli ja iga rikete tabeli jaoks rikete katte koondvektori.



Joonis 18. Testpunktide emuleerimine[10]

Igale testpunktile sünteesitakse talle vastav simuleerimistabel, mille põhjal Turbo-testri rikete simulaator arvutab rikete tabeli ja iga rikete tabeli jaoks rikete katte vektori. Programmi tulemusena tekitatakse Joonisel 18 esitatud testpunktide tabel.



Joonis 19. Testpunktide tabel

Testpunktide tabelis igale reale vastab rikete katte vektor, mille garanteerib antud reale vastava testpunkti sisse viimine skeemi.

3.2.4. Testpunktide efektiivsuse analüüs.

Selle programmi ülesandeks on tekitada eelmises punktis loodud testpunktide tabelist täpsustatud testpunktide tabel, kus igale testpunktile vastav rikete katte vektor arvutatakse ümber arvestades selle koosmõju algse rikete katte vektoriga, mis arvutati punktis 1. Nii saadakse iga testpunkti reaalne lisamõju ette antud funktsionaalsele testile.

Programmi sisendiks on rikete katte vektor antud testprogrammi jaoks (initTestedFaults) ja testpunktide abil saavutatavad rikete katte vektorid (cpLocal). Programmi väljundiks on aga uued rikete katte vektorid, mis on esitatud failidena cpLockalSub ja initTestedSub.

3.2.5. Unikaalsete testpunktide otsimine

a) Programmi sisendiks on testpunktide täpsustatud tabel, kus iga rida esitab rikete katte vektorit, mis saavutatakse sellele reale vastava testpunkti skeemi lisamise abil.
b) Programmi väljundiks on kokku liidetud unikaalsetele testpunktidele vastavate rikete katte vektorite tulemused ja selle tulemusel saadud kogu skeemi rikete kattuvus. Kontrolliks saab koostatud ka fail, milles on kõik esinenud unikaalsed testpunktid ja skeemi kattuvus pärast igat liidetud vektorit.
Unikaalseks nimetatakse testpunkti, millele vastavas rikete katte vektoris on vähemalt 1 sõlm, mille riket suudeti avastada ainult selle testpunkti lisamise korral. Unikaalse testpunkti näitena on Joonisel 20 esitatud testpunktide täpsustatud tabel, kus ainsale unikaalsele testpunktile vastab reavektor 34, kus eksisteerib veerg (skeemi sõlm), milles riket on võimalik avastada vaid üheainsa (selle sama unikaalse) testpunkti sisse viimise abil (testitud sõlm on märgistatud punase ringiga).

```
13: 1&1111111X11111XXX110011
28: 1&1&11111X11111XXX110011
34: 1&1&11&1X11111XXX110011
59: 1&1111111X11111XXX110011
```

Pilt 2. Unikaalne vektor

Joonis 20. Unikaalse testpunkti avastamine

Unikaalseid testpunkte avastav programmi kood on esitatud alljärgnevalt:

```
ArrayList<Integer> uniNod = new ArrayList<Integer>(nodsNotTested.size());
      //unikalne node


                  int vecNr=0; //testitava vektori number

                  int testN;  //testitava sõlme asukoha number

                  for(int n=0;n<nodsNotTested.size();n++)

                  {
```

```
                        testN=0;
                        for(int i=0;i<cpLocalSub.length; i++)
                        {
                              if(cpLocalSub[i].get(n)==FaultVector.AL)
                                    testN++;
                        }
                        writer3.println("X times to '&' = "+testN +" node
nr "+n );
                        if(testN==1){
                              uniNod.add(n);
                              writer5.println("Unique vec nr: "+vecNr+"
"+cpLocalSub[vecNr]);
                        }
                        //writer5.println("Unique vec nr: "+vecNr+"
"+cpLocalSub[vecNr]);
                        vecNr++;
                  }
```

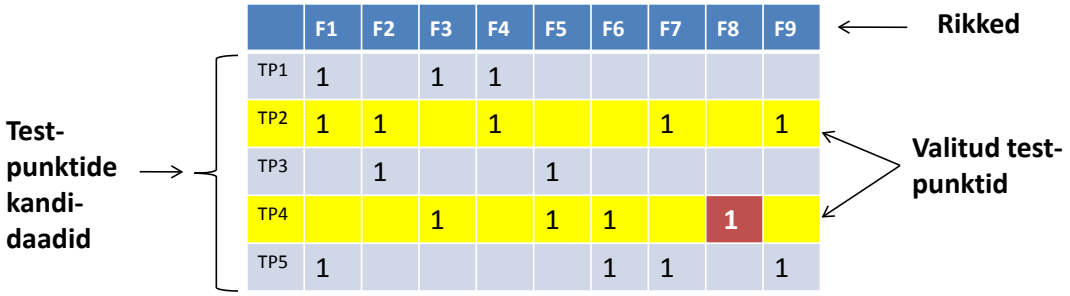
3.2.6. Testpunktide tabeli katte leidmine

Viimane programm arvutab lisaks unikaalsetele testpunktidele ülejäänud testpunktid (vajadusel), saavutamaks testpunktide abil rikete katte 100% etteantud testprogrammi jaoks.

a) Sisendiks vajab programm esmalt kõiki unikaalseid testpunkte ja neile vastavat summaarset rikete katte vektorit ning ka kõiki teisi analüüsitud testpunkte ning neile vastavaid rikete katte vektoreid „cpLocalSubCur[i] = new FaultVector(cpLocalSub[i])“

b) Programmi väljundiks saab skeemi täielikku 100%-list kattuvust garanteeriv testpunktide loetelu.

Programm töötab greedy algoritmi põhimõttel, otsides igal sammul välja parima riketekattuvusega seotud testpunkti, millega parandab kõige enam skeemi rikete kattuvust. Kui parima testpunkti lisamisel saavutatud üldine rikete kate on endiselt alla 100%, sooritatakse sama funktsioon uuesti, kuni on saavutatud rikete kattuvus 100% või on vajalike testpunktide nimekirja lisatud kõik võimalikud testpunktid analüüsitavast testpunktide täpsustatud tabelist.



Joonis 21. Testpunktide tabeli katte leidmist

Testpunktide tabeli katte leidmist illustreerib Joonis 21, millel on esitatud testpunktide tabel. Tabeli read vastavad testpunktidele ja veerud riketele, milliste avastamist testpunktide toetavad. Tabelis on 1, kui reale vastav testpunkt soodustab veerule vastava rikke testimist. Kõigepealt leitakse unikaalsed testpunktid (nt TP4), kus leidub veerg üheainsa 1-ga (nt rike F8). Seejärel leitakse ülejäänud testpunktid, nii et kõik rikked oleksid vähemalt ühe korra kaetud.

4. Eksperimentaalne uurimistöö

Eksperimentaalse uurimistöö eesmärgiks oli võrrelda kahte digitaalskeemide funktsionaalse isetestimise kvaliteeti. Tõstmise meetodit ette antud funktsionaalse testprogrammi poolt genereeritavate testvektorite hulga jaoks:

- testpunktide sisseviimine skeemi juhitavuse parandamiseks ja
- hübriidtestimise põhimõtte kasutamine, kus funktsionaalset isetesti täiendatakse mälus hoitavate deterministlike testvektoritega.

Testpunktide kasutamise uurimisel skeemi juhitavuse parandamiseks on aluseks võetud magistritöös väljatöötatud uus testpunktide minimeerimise meetod ja uus testpunktide imiteerimise meetod rikete katte analüüsimisel.

Meetodite võrdlemisel on kasutatud rahvusvaheliselt tunnustatud ISCAS'85 perekonna katseskeeme. Meetodite efektiivsust on võrreldud ka skaleeruvuse seisukohalt: on analüüsitud samu skeeme kahel keerukuse juhul, suurema keerukusega loogikaelementide tasemel ja väiksema keerukusega makrotasemel, kus makrodena vaadeldakse puukujulisi alamskeeme.

Testvektorite etteandmiseks on kasutatud kahte võimalust: (pikki ning liiaseid) pseudojuhuslikke testvektorite jadasid ja kompaktseid deterministlikke testvektorite jadasid. Testvektorite etteandmisel on uuritud testide kvaliteedi parandamise efektiivsust erinevate etteantud testide algkvaliteetide korral. Niisugusteks algkvaliteetideks on võetud rikete katted: 95%, 90%, 85% ja 80%.

Alljärgnevalt on antud lühiülevaade koos kompaktse esitusega ja kokkuvõetud järeldustega läbi viidud eksperimentidest. Ülevaade kõigist eksperimentidest on ära toodud Lisas.

4.1. Meetodite võrdlus testimise kvaliteedi parandamisel eri katseskeemide puhul

Võrreldi kahte meetodit: testpunktide kasutamine ja deterministlike lisavektorite kasutamine. Meetodid realiseeriti 6-l ISCAS'85 katseskeemil: c880, c1355, c1908, c3540 ja c6288. Tulemused on toodud Tabelis 1.

Tabel 1

	ISCAS'85 benchmark circuits					
	c880	c1355	c1908	c3540	c5315	c6288
Konstantset rikked	1550	2194	2788	5568	8638	9728
Liiased rikked	0	8	9	147	60	51
PR Lisavektorid/100	2694	1438	4420	9631	1793	42
PR lisavektorid/95	240	478	1457	1800	180	10
Det. lisavektorid/95	22	21	22	12	80	15
Testpunktid/95	3	4	10	112	54	30

Tabeli esimeses reas on iseloomustatud skeemide keerukust võimalike konstantsete rikete arvuga skeemis, mis võrdub 2-kordse ühenduste arvuga loogikaelementide võrgus. Teises reas on märgitud liiaste rikete arv skeemis, mida polegi võimalik testida. Antud eksperimentide grupis on kasutatud üksnes pseudojuhuslike algteste. Kolmandas reas on esitatud testi algpikkus, mis garanteerib 100%-lise rikete katte mitteliiaste rikete suhtes. Neljas rida annab ette algtesti, mille kvaliteeti iseloomustab 95%-line rikete kate. Viiendas reas on näidatud, kui palju deterministlikke lisavektoreid on vaja, et jõuda parima rikete katteni. Ja kuues rida näitab testpunktide vajadust testi kvaliteedi viimiseni 100%-ni.

Oluline on siin märkida, et kui deterministlike vektorite lisamine ei võimalda liiaste rikete avastamist, siis testpunktide sisseviimine seda võimaldab, mis on testpunktide meetodi lisaväärtuseks. Kui aga võtta eesmärgiks ainult mitteliiaste rikete avastamist, siis väheneks vajalike testpunktide arv märgatavalt. Kui palju see väheneks, sellele

käesolev eksperimentide seeria vastust ei anna. Selle probleemi uurimine võiks jääda tulevasteks uuringuteks.

4.2. Meetodite võrdlus sõltuvalt algtesti kvaliteedist

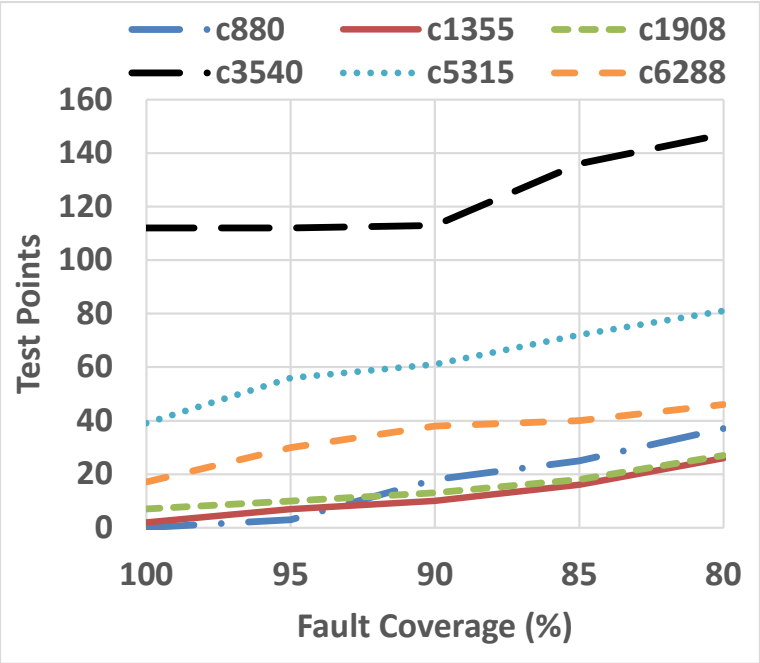
Järgmised katsed olid suunatud sellele, kuidas muutuvad testpunktide ja täiendavate *deterrministlike* testvektorite vajadus sõltuvalt algtesti kvaliteedist. Tabelites 2 ja 3 on esitatud andmed kahe katseskeemi c880 (Tabel 1) ja c3540 (Tabel 2) jaoks, kusjuures esimene on suhteliselt hästi testitav skeem, milles liiased rikked puuduvad, aga teine on raskesti testitav skeem, kus liiaste rikete hulk võrdub 147-ga (vt. Tabel 1).

Tabel 2

Vigade katteuvus, %	Esialgne	95	90	85	80
Juhuslikud lisavektorid	2694	240	114	73	33
Deternimistlikud lisavektorid	0	22	31	43	56
Testpunktid	0	3	18	25	37

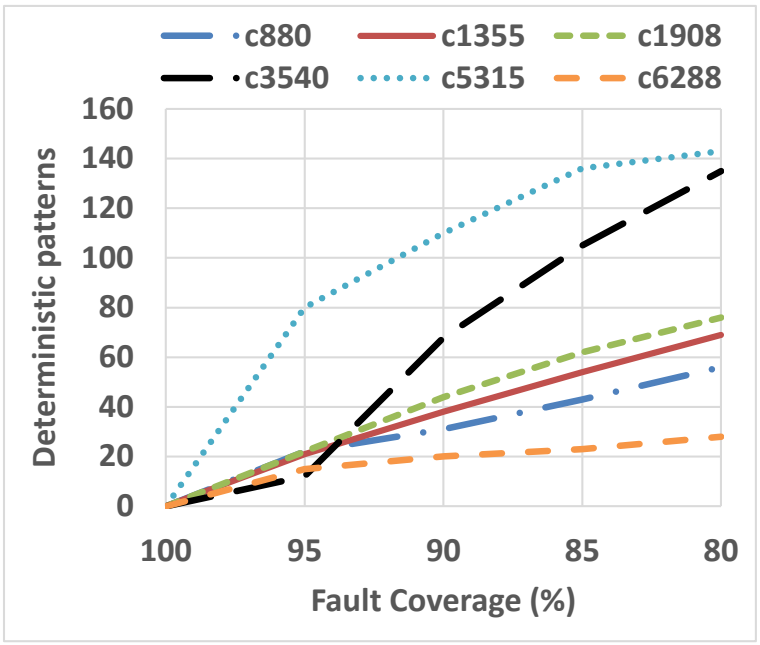
Tabel 3

Vigade katteuvus, %	Esialgne	95	90	85	80
PR lisavektor	9631	1800	414	161	94
Det. lisavektor	0	12	68	105	135
Testpunktid	112	112	113	136	147



Joonis 22. Testpunktide vajadus sõltuvalt algtesti kvaliteedist

Meetodeid on võrreldud tabelites 2 ja 3 testide algkvaliteetide 100%, 95%, 90%, 85% ja 80% puhul, vastavalt vasakult paremale. 100% line test skeemi c880 puhul ei vaja testpunkte, mis on ka arusaadav. Küll aga vajab neid skeem c3540, milles on palju liiaseid rikkeid. 112 juhtpunkti lisamine võimaldaks vajadusel ka need liiased rikked ära testida.



Joonis 23. Deterministlike testvektorite vajadus sõltuvalt testi algkvaliteedist

Joonised 22 ja 23 toovad võrdlevalt kõigi kuue katseskeemi jaoks graafiliselt esile, vastavalt, testpunktide ja deterministlike testvektorite vajaduse, sõltuvalt testi algkvaliteedist. On näha, et mida keerulisem ja halvemini testitav on skeem (c3540 – 147 liiast riket, c5315 – 60 liiast riket), seda kiiremini kasvab vajalike deterministlike testvektorite arv, seejuures kiiremini kui testpunktide arv.

Viimane tähelepanek rõhutab testpunktide kasutamise suuremat efektiivsust, juhul kui testide algkvaliteet on väike.

4.3. Meetodite võrdlus sõltuvalt skeemi keerukusest (makrotase vs. loogikaelementide tase)

Huvitav tähelepanek järgneb Tabelist 4, kus võrreldakse skeemi keerukuse mõju katseskeemi c880 näitel nii deterministlike testvektorite kui ka testpunktide lisamisel. Keerukusel on siin kahetine mõju:

- mida väiksem on mudeli keerukus, seda kiiremalt on võimalik mudelil läbi viia simuleerimist
- mida väiksem on mudeli keerukus, seda väiksem on ka modelleeritavate rikete arv (loogikatasemel on vaja modelleerida 1550 riket, makrotasemel vaid 994 riket).

Viimasest väitest võib arendada hüpoteesi, et ka täiendavaid testpunkte ja testvektoreid võiks makrotasemel vähem vaja minna võrreldes loogikaelementide tasemega.

Tabel 4

Vigade kattuvus, %		Esialgne	95	90	85	80
Gate level (1550)	PR lisavektorid	2694	238	106	48	28
	Det. lisavektorid	0	24	36	58	62
	Kontrollpunktid	0	4	15	21	30
Macro level (994)	PR lisavektorid	2694	240	114	73	33
	Det. Lisavektorid	0	22	31	43	56
	Kontrollpunktid	0	3	18	25	37

Tabelist 4 näeb, et kui algtesti kvaliteet on suhteliselt kõrge (nt. rikete kattega 95%), siis vajalike testpunktide arv on väga väike, sõltumata keerukuse tasemest, samas kui deterministlikke testvektoreid läheb palju vaja, nii loogikaelementide kui ka makrotasemel läbi viidud analüüsi tulemusena. Kui aga algtesti kvaliteet on madal (alla 95%), siis on nii testpunkte kui ka täiendavaid testvektoreid mõlemaid palju vaja.

Ülal formuleeritud hüpotees ei tööta testpunktide puhul, küll aga töötab testvektorite kasutamise korral.

4.4. Testpunktide arvu minimeerimine parameetri *k* suurendamise abil

Käesolevas töös läbi viidud eksperimentide puhul tehti tähelepanek, et vajalikku testpunktide arvu on üldkokkuvõttes võimalik vähendada, neid skeemi viies mitte üksnes nendes kohtades, kus rikked on jäänud avastamata, vaid ka nendes kohtades, kus rikete avastamise kordade arv testi käigus on väike.

Nimetatud tähelepanek juhtis võimalusele ehitada testpunktide tabelit rikete hulkade $R(k)$ põhjal, kus $k > 0$, ja mitte üksnes juhtumil, kus $k = 0$. Eksperimentide tulemused kinnitasid hüpoteesi, mis ennustas, et parameetri k suurendamise korral väheneb (või jääb endiseks) vajalik testpunktide arv. Parameetri k väärtus tähendas maksimaalset rikete avastamise arvu antud testi ajal, millal rikkega seotud skeemipunkt veel loetakse testpunkti sisseviimise kandidaadiks.

Tabel 5

Skeem	Esialgne vigade kattuvus %	Limit väärtusena *k* in *S*(*k*)st				
		0	1	5	10	20
c880	95	19	6	6	6	3
	90	23	20	21	18	18
	85	33	27	27	26	25
c1908	95	29	27	18	12	10
	90	36	37	19	19	13
	85	64	50	23	18	18
c3540	95	119	119	118	115	112
	90	202	185	148	139	113
	85	243	208	172	143	136

Tabelis 5 on selgelt näha, kuidas parameetri *k* kasvades väheneb (või jääb samaks) vajalike testpunktide arv 100%-lise rikete katte saavutamiseks. Näiteks skeemi c880 puhul osutus võimalikuks vähendada testpunktide arvu koguni 19-lt kuni 3-ni. Parameetri k effekt väheneb, kui testide algkvaliteet on madal.

Kokkuvõte

Käesoleva magistritöö tulemused võiks kokku võtta järgnevalt:

1. Magistri töö käigus töötati välja algoritmid, programmeeriti ja katsetati kahte uut meetodit testpunktide arvu minimeerimiseks skeemide juhitavuse parandamisel funktsionaalse isetestimise kvaliteedi tõstmise eesmärgil, kus testide algkvaliteet on ette antud:
    - Esimene meetod oli suunatud otseselt testpunktide arvu minimeerimisele, mis põhines parameetri *k* sobiva suuruse valikul, ehk siis limiidi määramisel testpunktide kandidaatide valikuks;
    - Teine meetod seisnes uudse võimaluse loomises testpunktide mõju analüüsiks, kus selle asemel, et modifitseerida skeemi ennast, modifitseeritakse skeemi simuleerimise tabelit, mis on lihtsam ja kiirem kui muutuste sisse viimine disaini.
2. Viidi läbi eksperimentaalne uurimistöö kahe meetodi võrdlemiseks funktsionaalse isetestimise kvaliteedi tõstmise eesmärgil digitaalskeemides, kus esimene meetod oli seotud testitavuse parandamisega täiendavate testpunktide sisseviimise abil ja teine täiendavate deterministlike testvektorite lisamisega hübriid-funktsionaalsel isetestimisel.
3. Huvitav on märkida, et esialgne tööülesanne seisneski vaid kahe juba tuntud meetodi võrdlemises ja uued meetodid sündisid just eksperimentaalse uurimistöö käigus, võimaldades eksperimentide läbiviimise efektiivsust ja ka tulemuslikkust tõsta.

Kasutatud materjalid.

1. L.-T.Wang, C.-W.Wu, X.Wen. VLSI test principles and architectures. Morgan Kaufmann, 2006.
2. S.Kostin, R.Ubar, G.Mägi. Comparison of two approaches to improve functional BIST fault coverage Baltic Electronics Conference. 2014. Submitted.
3. R.Ubar, S.Kostin, H.Kruus, M.Aarna, S.Devadze. Environment for Analysis of Functional Self-Test Quality in Digital Systems. Proc. of the Estonian Academy of Sciences, 2014, 63, 2 (to be published).
4. N.A.Touba, E.J.McCluskey. Test point insertion based on path tracing. Prov. VLSI Test Symp., 1996, pp.2-8.
5. Turbo Tester v02.10.[Manual]
6. Turbo Testeri kirjeldus. [WWW]http://www.pld.ttu.ee/magister/priidu.pdf (14.04.2014)
7. http://www.pld.ttu.ee/testing/theory/fault.html#structural
8. http://ati.ttu.ee/~artur/papers/TurboTester-EWDTC03.pdf (02.05.2014)
9. David Bryan. The ISACS ’85 benchmark circuits and netlist format.
10. Raimund Ubar, Õppeaine „Testide projekteerimine“

Lisa 1

Eksperimentaalse uurimistöö tabelid

Eksperimentide eesmärgiks oli võrrelda kahte funktsionaalse testi parandamise meetodit: deterministlike testide lisamist ja testpunktide lisamist. Võrdlus viidi läbi erinevate ette antud funktsionaalse testi kvaliteedi (rikete katete) juures. Algseks rikete katteks valiti väärtusi alates 60% kuni 95%. Võrdluseks võeti seejuures testide struktuur ka 100% juures (liiaste skeemide puhul võis see protsent ka väiksem olla).

Lisas on esitatud läbi viidud eksperimentide tabelid katseskeemide ISCAS'85 c3540 ,c5315, c1355, c6288, c1908, c880, c499 jaoks. Igat katseskeemi analüüsiti kahel keerukuse tasandil – loogikaelementide ja makrode tasandil, kusjuures makrode all on mõeldud puukujulisi aseskeeme.

Enamikel katseskeemidel on palju liiaseid rikkeid. Liiaste rikete mõju elimineerimiseks, on Arvutitehnika instituudis sünteesitud kõikide ISCAS'85 katseskeemide mitteliiased versioonid, mis võimaldasid saada 100% rikete katte. Ka nende skeemide suhtes viidi läbi samasugused eksperimendid. Sünteesitud katseskeemide tähistena on kasutatud tähte d (d3540 jne.).

Selleks, et uurida testide pikkuste mõju katsetulemustele võeti vaatluse alla kahel põhimõttel sünteesitud testid – deterministlikud testid (lühikese pikkusega testid) ja pseudojuhuslikud testid (pikad testjadad).

Lisamärkus: Veerus 4 „Lisatud deterministlike vektorite arv" on mõneskohas lisatud veel kaks parameetrit: / liiaste rikete arv ja // testimata jäänud rikete arv.

Gate-level

c3540, juhuslikud testid, kogu rikete arv = 5468						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
95	5291	1800	11	1811	178	18
90	5019	399	60	459	202	19
80	4456	93	129	222	252	94
70	3915	48	161	209	252	136
60	3345	32	165	197	248	276
c3540, deterministlikud testid						
95,51	5318	148			112	17
90	2020	65			136	34
80	4475	36			256	55
70	3924	22			292	184
60	3356	13			322	347
d3540, juhuslikud testid, kogu rikete arv = 2762						
100	2762	18283	-	-	-	-
96,198	2657	5291	53/7A //58	149	31	0
91,238	2527	1883	70/9A //77	150	66	11
85,95	2374	191	93/20A//102	161	86	103
81,535	2252	67	119/20A//133	159	99	129
d3540, deterministlikud testid						
100	2762	126			-	-
95	2623	58			69	2
90	2500	37			116	6
85	2349	31			110	86
80	2214	24			99	205

Makro-level skeemid

C3540 juhuslikud testid, kogu rikete arv = 3296						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
95,51	3148	0			133	0
90,3	2979	132	67/99U/55A	199	172	8
80	2634	94	105/99U/55A	204	194	33
C3540, deterministlikud testid						
95,48	3142	148			112	9
94,96	3130	136			119	14
90	2967	72			186	15
85	2805	53			210	24
80	2642	40			262	36
D3540, juhuslikud testid, kogu rikete arv = 2762						
99,93	2760	162	126/2A//355655			-
95,3	2633	5291	53//14000	5344	31	0
91,23	2487	1883	80//14717	1963	66	11
85	2349	191	105//84365	399	86	103
80	2216	67	124/1A//180636	211	99	129
D3540, deterministlikud testid						
100	2762	124				
95,92	2624	58			69	2
90	2501	37			116	6
85	2350	31			110	86
80	2215	24			99	205

Gate-level

C5315, juhuslikud testid, kogu rikete arv = 8638						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
95	8287	140	75	215	125	36
90	7775	68	108	176	218	64
80	6949	30	123	153	304	192
70	6174	21	120	141	397	262
60	5343	13	127	140	420	464
C5315, deterministlikud testid						
99,282	8576	104			39	1
90	7793	25			248	1
80	6941	15			314	59
70	6951	10			400	132
60	5209	7			364	274
D5315, juhuslikud testid, kogu rikete arv = 8638						
100	5050	79	-	-	-	-
90	4578	22	95	117	7	0
80	4078	13	118	131	8	0
70	3609	9	121	130	18	0
60	3032	6	141	147	28	0
D5315, deterministlikud testid						
100	5050	96			-	-
95	4813	32			62	2
90	4578	22			95	16
85	4301	16			174	37
80	4077	13			210	55

Makro-level skeemid

C5315, juhuslikud testid, kogu rikete arv = 5424						
94,304	5115	75	72	197	151	1
89,915	4877	49	112/2A	161	194	9
80,7	4377	32	130/2A	162	249	92
70	3814	24	146/2A	170	294	220
60						
C5315, deterministlikud testid						
98,802	5359	104			39	1
95	5155	45			159	1
90	4899	30			231	27
85	4640	23			274	33
80	4416	19			284	49
D5315 juhuslikud testid, kogu rikete arv = 5050						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
100	5050		-		362	
95,18	4807	110	11/5A//300974	121	74	4
90	4550	74	25/6A//350224	99	115	8
85,089	4297	50	25/7A//400250	75	125	27
80	4051	34	33/5A//301771	67	171	42
D5315, deterministlikud testid						
100	5050	96			-	-
95	4813	32			92	3
90	4577	22			126	21
85	4301	16			174	37
80	4077	13			210	55

Gate-level

C1355 juhuslikud testid, kogu rikete arv = 2194						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
95	2089	475	23	498	28	0
90	1976	169	45	214	15	0
80	1756	60	72	132	27	3
70	1550	34	85	119	35	3
60	1403	29	89	118	37	3
C1355, deterministlikud testid						
99,134	2175	86			8	0
90	1976	37			59	0
80	1773	12			56	2
70	1600	7			84	34
60	1327	4			130	95
D1355, juhuslikud testid, kogu rikete arv = 1500						
100	1500	82	-	-	-	-
90	1355	28	53	81	8	0
80	1212	11	78	89	15	0
70	1072	7	84	91	23	0
60	997	5	88	93	29	0
D1355, deterministlikud testid						
100	1500	82			-	-
95	1423	51			28	0
90	1354	28			34	0
85	1281	16			44	1
80	1211	11			47	1

Makro-level skeemid

C1355 juhuslikud testid, kogu rikete arv = 1618						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
100(95)	1525	89	28/8A	117	30	0
90	1461	71	44/8A	115	43	1
80	1297	37	76/8A	113	58	9
C1355, deterministlikud testid						
99,134	1610	76			8	0
95	1537	65			34	0
90	1455	42			53	0
85	1376	24			47	0
80	1297	13			54	1
D1355 juhuslikud testid, kogu rikete arv = 1500						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
98,9	1484	67	98/15A		10	0
95,2	1428	429	24		19	0
90	1352	173	41/13A		23	0
85	1281	65	53/15A		20	0
80	1202	37	68/16A		24	0
D1355, deterministlikud testid						
100	1500	82				
95	1426	51			28	0
90	1355	28			34	0
85	1282	16			44	1
80	1212	11			47	1

Gate-level

C6288 juhuslikud testid, kogu rikete arv = 8638						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
99,259	9656	42	0/18A/62U	42	18	16
95	8207	140	15/18A/62U	155	38	16
90	7776	68	20/6A/62U	88	51	16
85	7346	43	22/14A/62U	65	62	16
80	6949	30	26/15A/62U	56	66	16
C6288, deterministlikud testid						
99,208	9651	22			18	16
95,27	9268	10			40	16
91,7	8921	8			62	16
88,4	8601	7			71	16
88,6	8234	6			86	14
D6288, juhuslikud testid, kogu rikete arv = 9438						
100	9438	30	-	-	-	-
95	8997	12	39	51	54	0
90	8504	8	44	52	106 / 58	2 / 1
85	8280	7	45	52	113	2
80	7984	6	46	52	128 / 101	7 / 6
D6288, deterministlikud testid						
100	9438				-	-
95,327	8997	12			54	0
90,125	8488	8			106	2
87,741	8281	7			113	2
84,605	7985	6			128	7
Makro-level skeemid C6288, juhuslikud testid, kogu rikete arv = 7744						
99,328	7692	42	0/48U/3A	42	18	16
95	7361	10	16/48U/3A	26	38	16
91,23	7065	8	21/48U/3A	29	51	16
85	6793	7	24/68U/3A	31	60	16
80	6485	6	29/48/3A	35	66	16
C6288, deterministlikud testid						
99,3	7690	22			18	16
95,92	7428	10			40	16
90	7026	7			62	16
87,46	6773	6			71	16
80	6342	5			86	14
D6288, juhuslikud testid, kogu rikete arv = 9438						
100	9438		-0	-	13	0-
95	8987	15	33//20	48	45	0
90,5	8541	10	46//34	56	93	1
85	8161	8	51//21	59	122	8
80	7847	7	46//49	53	132	10
D6288, deterministlikud testid						
100	9438	36			-	-
95,4	9009	12			30	0
90	8504	8			58	1
87,46	8780	7			113	2
80	7984	6			128	7

Gate-level

C1908 juhuslikud testid, kogu rikete arv = 2788						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
95,158	2653	4420	40/7A/6U	4860	8	0
95.01	2645	4041	42/7A/6U	4083	9	0
95,876	2673	1457	20/7A/6U	1477	23	0
89,02	2482	545	55/6A/6U	400	38	1
80	2328	240	76/6A/6U	316	39	5
C1908, deterministlikud testid						
96,34	2686	123			7	0
95	2650	102			9	0
90	2512	57			31	0
85	2372	34			49	13
80	2247	23			65	68
D1908, juhuslikud testid, kogu rikete arv = 1128						
100	1128	1135	-	-	-	-
95	1084	376	9/10A	385	9	0
90	1016	214	23/9A	237	23	0
85	986	188	24/11A	212	38	1
80	903	36	30/11A	66	39	5
D1908, deterministlikud testid						
100	1128	40			-	-
93,17	1151	24			16	0
87,77	990	19			18	0
82,36	929	13			26	2
80	907	9			37	2

Makro-level skeemid

C1908, juhuslikud testid, kogu rikete arv = 1732						
99,36	1721	4421	1/5U/4A	4423	8	0
90,01	1559	765	45/5U/5A	813	30	0
85	1473	408	62/5U/4A	470	46	1
80,2	1390	245	76/5U/4A	318	49	10
C1908, deterministlikud testid						
100	1723	62			7	0
95	1658	1457			27	0
90	1558	769			37	2
85	1478	408			50	15
80	1390	245			61	59
D1908 juhuslikud testid, kogu rikete arv = 1128						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
100	1128		-		18	1
96,1	1084	378	11/5A//300974	389	3	0
90	1016	214	25/6A//350224	239	6	0
85	986	188	25/7A//400250	213	7	0
80	903	36	33/5A//301771	69	20	6
D1908, deterministlikud testid						
100	1128	40			-	-
95	1081	24			16	0
90	1022	19			18	0
85	961	13			26	2
80	908	9			37	2

Gate-level

C880 juhuslikud testid, kogu rikete arv = 1550						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
100	1550	56	-	-	-	-
95	1473	21	33	54	35/20	3 / 0
85	1398	14	55	69	39/26	2 / 2
80	1277	10	60	70	42/30	3 / 3
C880, deterministlikud testid						
100	1550				-	-
92,12	1428	61			13	0
87,04	1349	46			20	0
85	1321	29			28	1
80	1241	22			29	11
D880, juhuslikud testid, kogu rikete arv = 1274						
100	1274	50	-	-	-	-
95	1211	22	27	49	23	1
90	1159	17	33	50	38	5
85	1096	13	38	51	46	6
80	1031	11	50	61	50	17
D880, deterministlikud testid						
100	1274	50			-	-
95	1165	22			22	1
90	1119	17			38	5
85	1095	13			46	6
80	1030	11			50	17

Gate-level

C499 juhuslikud testid, kogu rikete arv = 2194						
Rikete alg kate	Avastatud rikete arv	Testi pikkus	Lisatud det. vektorite arv	Testi kogupikkus	Testpunktide vajalik arv	Testimata rikked
99,1796	2176	1373	2/10A	1375	8	0
95	2086	534	20/10A	558	13	0
90	1977	167	43/8A	210	29	0
85	1867	79	61/8A	140	13	0
80	1770	48	71/8A	119	25	0
C499, deterministlikud testid						
99,088	2174	85			8	0
95	2085	63			32	0
90	1977	37			49	0
85	1866	20			44	0
80	1768	12			57	0
D499, juhuslikud testid, kogu rikete arv = 1560						
100	1550	80	-	-	-	-
95	1483	47	21 / 29A	68	31	0
90	1407	26	35 / 52A	61	32	1
85	1324	14	50 / 59A	64	49	2
80	1266	10	58 /56A	68	57	4
D499, deterministlikud testid						
100	1560	80			-	-
95	1483	47			31	0
90	1407	26			32	1
85	1324	14			49	2
80	1266	10			57	4

Lisa 2

Artikkel, mis on esitatud konverentile Baltic Electronic Conference, Oct. 4-6. 2014.

Comparison of two approaches to improve functional BIST fault coverage

Sergei Kostin, Raimund Ubar, Gunnar Mägi

***Abstract*— Two approaches to improve the fault coverage of functional logic BIST in digital circuits are proposed and investigated. The first approach is based on introducing of additional test points. An experimental tool set is developed for fast evaluation of the number of control points that is needed to achieve 100% fault coverage for the given functional test sequence. A novel algorithm is proposed to minimize the number of control points. The second approach is based on complementing the functional test with additional deterministic test patterns. The pros and cons of both approaches are highlighted and discussed.**

Keywords – functional BIST, hybrid BIST, design for testability, controllability, test point insertion

1 Introduction

Increasing size and complexity of digital circuits impose new challenges to testing modern chips, requiring accurate dynamic tests. Therefore testing of chips by so called at-speed test is becoming the must. The use of conventional scan chains has proven to be often inadequate increasing the cost in terms of additional hardware and testing time [1], excessive power dissipation during test [2] and leading to yield loss because of over-testing [3]. The burden of external testers has been relieved by introducing self-test approaches such as hardware-based logic Built-in Self-Test (LBIST) which typically use Linear Feedback Shift Registers (LFSR) [4]. The LBIST involves using on-chip hardware to apply pseudorandom test patterns to the Circuit Under Test (CUT). Established LBIST solutions use special hardware for test pattern generation (TPG) and test response evaluation (TRE) on chip, but this in general introduces significant area overhead and performance degradation. To overcome these problems, the methods of Functional BIST (FBIST) have been proposed which exploit specific functional units such as arithmetic units or processor cores for on-chip test pattern generation and test response evaluation [5-7]. In particular, it has been shown that adders can be used as TPGs for pseudo-random, pseudo-exhaustive and deterministic patterns. An important trend today is the at-speed test [8] having additional benefit of the ability to test circuits under conditions that are as close as possible to normal circuit operation.

The question is whether a self-test sequence running in the system can adequately exercise its hardware components satisfying the targeted fault coverage requirements. Unfortunately, many circuits usually contain Random-Pattern-Resistant (RPR) faults [9] which limit the fault coverage that can be achieved with the described approaches based on LBIST and FBIST.

There has been several methods developed for improving the fault coverage when using the methods referred above. One group of methods is to modify the CUT by either inserting test points [9, 10] or by redesigning it to improve the fault coverage [11-12]. Another group of methods is trying to improve the fault coverage by implementing “mixed mode” or hybrid BIST approaches where deterministic patterns are used to detect the faults that the pseudorandom patterns miss [13-16].

In this paper, we carry out an experimental comparison of two approaches to improve the fault coverage of functional logic BIST. The first approach is based on introducing of additional test points, and the second approach is based on complementing the functional test with additional deterministic test patterns. We will investigate, how the efficiency of these approaches is depending on the initial fault coverage achieved by the BIST in its natural pure way. We will also present a novel method to minimize the number of test points to achieve the 100% fault coverage.

2. Overview about state-of-the-art

The major challenge of LBIST or FBIST is the presence of RPR faults which have low detection probability by pseudo-random patterns, and, hence, may limit the achievable fault coverage. There are a lot of approaches for detecting RPR faults: modifying the test pattern generator, modifying the CUT by introducing test points, or using hybrid test approaches.

In the first case, weighted pseudorandom sequences are generated by LBIST [13, 18-21].

Additional logic is needed to weight the probability of each bit in the test sequence. The weight logic can be placed either at the input of the scan chain [18] or in the individual scan cells themselves [13, 19]. The disadvantage of the approach is in the need of storing the weight sets on chip and also, control logic is required to switch between these weights, so the hardware overhead may be large. A number of other techniques have been developed, such as pattern mapping [11], bit-fixing [9], bit-flipping [15], LFSR reseeding [13, 19, 22].

A very efficient method to cope with RPR faults is test point insertion [9,10], used either to improve the controllability or observability of internal signals in the circuit [9,10,17]. Since test points add area and performance overhead, an important issue for test point insertion is where to place them in CUT in order to maximize the fault coverage at minimum number of test points. Minimization of the number of test points was shown to be NP-complete [23], however, a number of approximate techniques have been proposed [9, 24].

"Mixed mode" approaches to BIST, called as well hybrid BIST, where deterministic patterns are used to detect the faults that the pseudorandom patterns miss, is another technique to improve the quality of embedded self-testing. However, storing deterministic patterns may require a large amount of hardware overhead. In [13], a technique based on reseeding an LFSR is proposed that reduces the storage requirements, and in [16] a mixed-mode approach is presented, in which deterministic test cubes are embedded in the pseudorandom sequence of bits itself. In [25], a hybrid functional BIST (HyFBIST) scheme is proposed where a trade-off between the on-line running functional test routine and the deterministic test patterns to be pre-generated and stored in the memory of the system is proposed.

In this work we present a case study to compare two approaches to improve the quality of self-test: by control point insertion to raise the testability of CUT on one hand, and generation of deterministic test patterns for HyFBIST, on the other hand. Both approaches assume that a functional test routine (FBIST) with a related fault coverage is given, and the task is to tailor either the control point set or deterministic additional testy patterns for the given FBIST sequence. To be better focused in this comparison, we will consider in this paper the case of using only the control points to improve the testability of the given circuit.

3 Control point insertion (CPI)

Consider a micro-programmed Arithmetic and Logic Unit (ALU) with a register block to store the intermediate results of micro-programs in Fig.1. All the micro-operations are carried out in the ALU which has the role of CUT. The ALU has data inputs and outputs connected via buses to the register block. The control signals from the control unit serve as additional inputs for ALU, and few status signals of the ALU serve as additional output signals connected to the control unit.

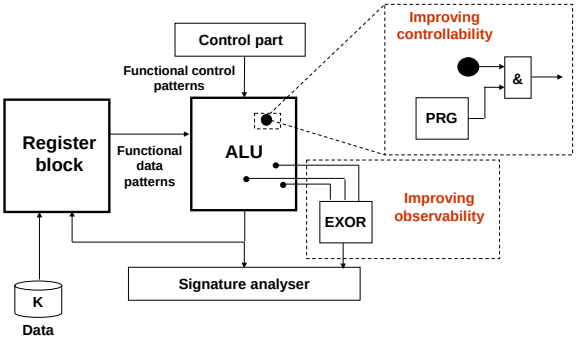


Fig.1. *A data path of a digital system with improved testability*

During n cycles of the given micro-program, which is used in the role of FBIST, the ALU can be exercised with n different functional patterns, and the responses of ALU can be compressed by the signature analyzer which is used to monitor the whole data manipulation process of the given micro-program. In this scheme the functional patterns produced directly on the inputs of ALU have the similar role as pseudorandom test patterns in classical LFSR based logic BIST schemes. Similarly to the pseudorandom test, the functional test patterns are not able to cover random-pattern-resistant faults, which limits the fault coverage that can be achieved with the pure functional BIST approach.

The testability of ALU can be improved by insertion of test points for improving either controllability or observability, or both. Insertion of observation points involve making a node observable by connecting it either directly or via EXOR circuit as shown in Fig.1 to signature analyzer. Insertion of control points involve either ANDing (as illustrated in Fig.1) or ORing a node with an activating signal. The activating signals typically are driven by pseudorandom values supplied by pseudorandom generator (PRG) during the FBIST session, and is set to a non-controlling value during normal operation.

The quality of the set of functional test patterns generated during the FBIST test sequence will be measured by fault simulation, and the random-pattern-resistant faults are determined.

To minimize the number of control points we propose the following novel method. By fault simulation of the given FBIST sequence T of test patterns we will determine the subsets of nodes $S(k)$ in the circuit, where k means that for each node $s \in S(k)$ the fault at this node is detected k or less times during the test T. If $k = 0$, then the fault undetectable by T. In this case, the node will be included into the set of candidate nodes for insertion of control points.

Denote this set by $S(0)$. To find the final set of nodes to be used as control points we may apply a similar method as in [9] based on the matrix M of test point solutions for each undetected fault. The final optimized set of control points $S_{min} \subseteq S(0)$ will be find by minimizing the coverage of the matrix M.

Consider a gate with fan-out branches at the output. Let for one of the inputs of the gate there is a fault which is detected k times by T. If $k > 0$, but still very small, then the probability of testing the faults at the output branches of the gate is expected to be the smaller, the smaller is k. This consideration suggests to include into the set of candidate nodes for insertion of control points not only the nodes of $S(0)$, but all the nodes of $S(k^*)$ where the value of k^* can be predetermined. In this case the matrix M will be created on the basis of the nodes in $S(k^*)$. The hypothesis is that the bigger is k^* the smaller will be the final set $S \subset S(k^*)$ of control points. The hypothesis was proved to be correct by experiments described in Section 5.

4 Hybrid functional BIST (HyFBIST)

Let consider the similar data path of a digital system as depicted in Fig.1, however, in the case of using the HyBIST approach we introduce a set of multiplexers at the input of ALU to have the possibility of selecting the source of test patterns, either using the normal working mode, or using the stored in the memory additional deterministic test patterns, as shown in Fig.2.

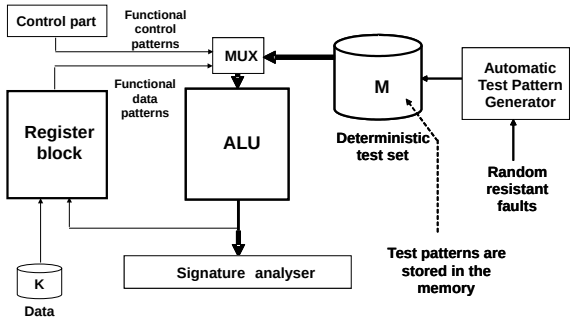


***Fig.2**. Functional BIST with adding deterministic test patterns*

The hybrid functional test session is carried out in two phases. In the first phase, the micro-program (as a part of the functionality of the system) is used to control the data-path based on some deterministic or random input data (operands) as the arguments of the micro-program. For example, in case of the multiplication micro-program, the two operands to be multiplied are used as the arguments. The micro-program can be repeated several times with different arguments to achieve a better fault coverage. The trade-off possibilities between the length of the FBIST micro-program (the number of repetitions) was investigated in [26]. In this paper, the target will be not to find a tradeoff between the lengths of the functional and deterministic parts of the test, rather to find the number of needed deterministic test patterns at the given fault coverage of the functional part of the test.

5. Experimental comparison of CPI and HyFBIST

Experimental results of comparison of the two approaches are presented in Tables 1-5 and in Fig. 3.

Table 1. Comparison of two approaches: CPI vs. HyFBIST

	ISCAS'85 benchmark circuits					
	c880	c1355	c1908	c3540	c5315	c6288
Stuck-at Faults	1550	2194	2788	5568	8638	9728
Collapsed faults	994	1618	1732	3296	5424	7744
Redundant faults	0	8	9	147	60	51
PR patterns/100	2694	1438	4420	9631	1793	42
PR patterns/95	240	478	1457	1800	180	10
Det. patterns/95	22	21	22	12	80	15
Contr. points/95	3	4	10	112	54	30

The results of experiments with 6 different ISCAS'85 circuits are presented in the last two rows of Table 1. As the basis of the functional part of the FBIST, the pseudorandom test patterns were used. The length of the test "PR patterns/100" means the number of patterns that was needed to cover all the detectable faults. Then the length of the test was reduced, to have the 95% fault coverage. For this level of initial FBIST quality, the needed additional deterministic patterns and control points were calculated, and the results are depicted in the last two rows. However, these results need different interpretation. Whereas the number of deterministic patterns for HyBIST were generated to achieve the 100% fault coverage regarding the testable faults, the number of control points were calculated to detect as well all the redundant (untestable) faults. This explains the high number of needed control points for the circuits in the last three columns where the number of redundant faults is very high.

Table 2. Experiments with c880

Fault coverage, %	Initial	95	90	85	80
PR patterns	2694	240	114	73	33
Det. patterns	0	22	31	43	56
Control points	0	3	18	25	37

Table 3. Experiments with c3540

Fault coverage, %	Initial	95	90	85	80
PR patterns	9631	1800	414	161	94
Det. patterns	0	12	68	105	135
Control points	112	112	113	136	147

In Table 2 and 3, for two circuits, the numbers of needed deterministic patterns and test points are shown as the function of the level of quality of the initial FBIST. Here we see a possibility of trade-offs between the FBIST test length and the improvements of the test quality either by deterministic patterns or control points. In Fig. 3 the same dependence is shown for all 6 circuits.

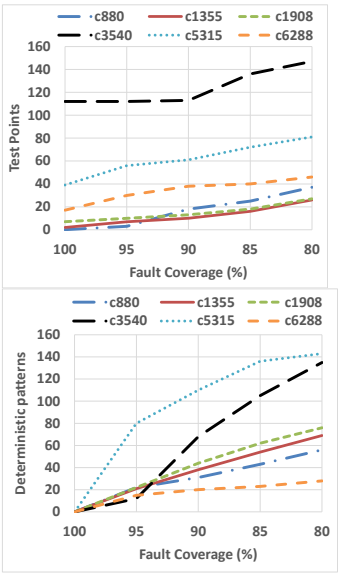


Fig.3 *Comparison of the approaches of CPI and adding deterministic test patterns to improve the fault coverage of FBIST*

In Table 4, the dependency of the results on the model complexity of the circuit is demonstrated. The complexity is less in the macro-level model where only the inputs of the fan-out free regions in the circuit are represented as nodes in the model. The complexity of the models is characterized by the number of faults shown in the 1st column in parentheses, whereas the numbers of nodes in the models is twice less. The results are shown for the circuit c880 only, however, a similar tendency was discovered for all other circuits as well. The reduction in complexity (in the number of nodes) increases the number of control points which can be explained by worse flexibility in placing the control points. The difference increases when the number of needed control points is increasing. In case when this number is small, even the opposite may happen (the column of fault coverage 95%).

On the other hand, the number of additional deterministic test patterns will be lower in case of the less complex model. This tendency can be explained by the fact that in average the ability of a test pattern to cover the faults is higher than the same for control points.

An interesting additional finding follows from Table 4 regarding the comparison of CPI vs. HyFBIST. If the initial quality of FBIST is high (95%), CPI seems to be more cost efficient than adding deterministic patterns (3 control points vs. 22 patterns). On the other hand, if the initial quality of FBIST is low (80%), the opposite is valid (37 control points vs. 56 patterns). However, the exact trade-off can be decided only when the real costs of CPI and HyFBIST are available.

Table 4. Comparison of gate and macro level analysis for c880

Fault coverage, %		Initial	95	90	85	80
Gate level (1550)	PR patterns	2694	238	106	48	28
	Det. patterns	0	24	36	58	62
	Control points	0	4	15	21	30
Macro level (994)	PR patterns	2694	240	114	73	33
	Det. patterns	0	22	31	43	56
	Control points	0	3	18	25	37

Table 5. Comparison of the effect of limits for fault detectability

Circuit	Initial fault coverage %		Limit for k in $S(k)$			
		0	1	5	10	20
c880	95	19	6	6	6	3
	90	23	20	21	18	18
	85	33	27	27	26	25
c1908	95	29	27	18	12	10
	90	36	37	19	19	13
	85	64	50	23	18	18
c3540	95	119	119	118	115	112
	90	202	185	148	139	113
	85	243	208	172	143	136

In Table 5, the efficiency of the new method of minimization of the control points is illustrated. The bigger is k (initial number of candidate control points), the less will be the final number of selected control points. For example, in the case of the circuit c880, we need in total to place 19 control points if we target to control only the nodes where the faults are not detected. If we increase the limit k up to 5, we need only 6 control points, and if we put k = 20, we need to place only 3 control points. The numbers of control points for the circuit c3540 are high for the same reason of the high number 147 of redundant faults as explained at the Table 1.

6 Conclusions

In this paper, the contribution is twofold. First, the goal was to compare two approaches to improve the testing quality of the functional BIST where the working routines (programs or micro-programs) are used as self-test sequences. Several interesting findings were explained in Section 5 regarding experimental results.

The second contribution is a novel method for minimization of control points where we introduced a limit for creating of the initial set of candidates for selecting the control points. We showed experimentally that the bigger will be the limit k, the better results we may achieve when trying to minimize the number of control points.

Acknowledgments

This work was jointly supported by EU through European Regional Development Fund, and FP7-2013-ICT-11: 619871 project BASTION as well as by the institutional research funding IUT 19-1 of the Estonian Ministry of Education and Research.

References

[1] L.Bushard, N.Chelstrom, S.Ferguson, B.Keller. DFT of the Cell Processor and its Impact on EDA Test Software. In IEEE Asian Test Symposium, 2006, pp. 369-374.

[2] S.Wang, S.Gupta. ATPG for heat dissipation minimization during scan testing. In ACM IEEE DAC, 1997, pp. 614-619.

[3] L.Chen, S.Ravi, A.Raghunathan, S.Dey. A Scalable Software-Based Self-Test Methodology for Programmable Processors. In IEEE/ACM Design Automation Confernce, 2003, pp. 548-553.

[4] L.-T.Wang, C.-W.Wu, X.Wen. VLSI test principles and architectures. Morgan Kaufmann, 2006.

[5] R. Dorsch, H-J. Wunderlich: Accumulator Based Deterministic BIST. ITC, 1998,Washington D.C.

[6] J. Rajski, J. Tyszer. Arithmetic BIST For Embedded Systems, Prentice-Hall, N J (1998).

[7] S. Chiusano, S. Di Carlo, P. Prinetto, H.-J. Wunderlich: On applying the set covering model to reseeding. Proc. DATE, 2001, pp. 156-161

[8] B.Nadeau-Dostie. Design For At-Speed Test, Diagnosis and Measurement. Kluwer Academic Publishers, 2002.

[9] N.A.Touba, E.J.McCluskey. Test point insertion based on path tracing. Prov. VLSI Test Symp., 1996, pp.2-8.

[10] N.Tamarapalli, J.Rajski. Constructive multi-phase test point insertion for scan-based BIST. Proc. ITC, Oct. 1996, pp.649-658.

[11] M.Chatterjee, D.K.Pradhan, W.Kunz. LOT: Logic optimization with testability - New transformations using recursive learning. Proc. Int. Conf. on CAD, Nov. 1995, pp.318-325.

[12] Z.Zhao, B.Pouya, N.A.Touba. BETSY: Synthesizing Circuits for a specified BIST environment. Proc. ITC, Oct. 1998, pp.144-153.

[13] B.Koenemann. LFSR-coded test patterns for scan designs. Proc. European Test Conf., Mar. 1991, pp.237-242.

[14] S.Hellebrand et. al., Built-in test for circuits with scan based on reseeding of ,ulti-polynomial linear feedback shift registers. IEEE Trans. On Comput. Vol. 44, pp.223-233, Feb. 1995.

[15] H.-J. Wunderlich, G.Kiefer. Bit flipping BIST. Proc. ICCAD, Nov. 1996, pp.337-343.

[16] N.A. Touba, E.J.McCluskey. Bir-fixing in pseudorandom sequences for scan BIST. IEEE Trans. on CAD of IC and Systems, Vol.20, No.4, Apr.2001.

[17] J.-S.Yang, B.Nadeau-Dostie, N.A.Touba, "Test point insertion using functional flip-flops to drive control points," ITC, 2009, pp. 1-10.

[18] M.F. AlShaibi, Ch.Kime. MFBIST: A BIST method for random pattern resistant circuits. Proc. ITC, Oct. 1996, pp.176-185.

[19] S.Hellebrand, J.Rajski, S.Tarnick, B.Courtois, S.Venkataraman. Built-in test for circuits with scan based on reseeding of multi-polynomial LFSR. IEEE Trans. on Comput. Vol. 44, pp.223-233, Feb. 1995.

[20] A.Jas, C.Krishna, N.A.Touba, "Hybrid BIST based on weighted pseudo-random testing: a new test resource partitioning", VTS, 2001, pp.2-8.

[21] L.Lai, et al., "HW efficient LBIST with complementary weights," Int. Conf. on Computer Design, 2005, pp.479-481.

[22] J.Rajski et. al, "Embedded deterministic test for low cost manufacturing test," ITC, 2002, pp. 301-310.

[23] B. Krishnamurthy, "A dynamic programming approach to the test point insertion problem," 24th DAC, 1987, pp.695-704.,

[24] M.Youssef. Y.Savaria, B.Kaminska, "Methodology for efficiently inserting and condensing test points," IEEE Proc. Computerss and Digital Techniques, vol.140, pp.145-160, May, 1993.

[25] R.Ubar, N.Mazurova, J.Smahtina, E.Orasson, J.Raik. HyFBIST: Hybrid Functional Built-In Self-Test in Microprogrammed Data-Paths of Digital Systems. Int. Conference MIXDES, Szczecin, June 24-26, 2004, pp.497-502.

[26] R.Ubar, Viljar Indus, Oliver Kalmend, T.Evartson. Functional Built-In Self-Test for Processor Cores in SoC. The 30th IEEE NORCHIP Conference, Copenhagen, Denmark, Nov. 12-14, 2012