TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAF40LT

Andreas Rosenfeld 134209IASB

DIGITAALSÜSTEEMI FUNKTSIONAALSE ISETESTIMISE VÕIMEKUSE UURIMINE

Bakalaureusetöö

Juhendaja: Raimund-Johannes Ubar
PhD
professor

Tallinn 2016

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.

Autor: Andreas Rosenfeld

23.05.2016

Annotatsioon

Antud töös uuritakse digitaalsüsteemi funktsionaalse isetestimise võimekuse hindamist ühe kindla algoritmi piirides, mis põhineb funktsioonil y=(a+b)(a-b). Põhieesmärgiks on võimalikult hea rikete katteprotsendi saavutamine ning kitsaskohtade väljaselgitamine, mis ei luba saavutada 100% rikete katet.

Selle saavutamiseks luuakse spetsiaalne programm, mis genereerib algoritmi töö tulemusena tsükliliselt tekkivaid ALU sisendjadasi. Neid testfaile analüüsitakse digitaalsüsteemide diagnostikatarkvaraga Turbo-Tester.

Testimise tulemusena saavutati algortimi 12. kordse jooksutamisega 87,75% rikete kate. Arvestades kasutatud algoritmi omadust, mis nõuab, et osa sisendeid on konstantsed, võib järeldada, et saavutatud tulemus on antud ülesande juures maksimumilähedane.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti kahekümne neljal leheküljel, kuute peatükki, kaheksat joonist, kolme tabelit.

Abstract

Study on the capability of functional self-testing of a digital system

Every day we use electronical devices that contain complex digital technology. Faults in those systems can be minor – a electronical thermometer stops working – or highly dangerous – a planes autopilot control system malfunctions. Those systems must undergo tesing to avoid serious accidents that are caused by faults in digital systems in the future.

In this thesis we analyze the capability of functional self-testing of a digital system. The desinged ALU is constructed using a algortihm that operates with the function y=(a+b)(a-b). Also a special program is written that generates the needed inputs every cycle of the used algorithm. These tests are then fed to a digital systems diagnostics software Turbo-Tester to determine the fault coverage.

The constructed tests showed us that we can manage a decent fault coverage of slightly under 90% (87,75%). This is achieved with combination of 12 seperate runs of the used algorithm. This is due to the characteristics of the used algorithm, that requires some of the input registers to be in a constant state.

The thesis is in estonian and contains twenty-four pages of text, six chapters, eight figures, three tables.

Lühendite ja mõistete sõnastik

ALU	*Arithmetic Logic Unit*, aritmeetika-loogikaplokk
ATPG	*Automatic Test Pattern Generator*, automaatne testide generaator
BIST	*Built-In Self-Test*, sisseehitatud isetestimine
CUT	*Component Under Test*, testitav skeem
FBIST	*Functional BIST*, funktsionaalne sisseehitatud isetestimine
INV	inversioon
LFSR	*Linear-Feedback Shift Register*, lineaarse tagasisidega nihkeregister
MISR	*Multiple Input Signature Analyzer*, mitme sisendiga signatuuranalüsaator
MUX	multiplekser
PRG	*Pseudo-Random Generator*, pseudo-juhuslik generaator
Rg	register
SA	*Signature Analyzer*, signatuuranalüsaator
s-a-0	*Stuck-at-0*, 0-konstantrike
s-a-1	*Stuck-at-1*, 1-konstantrike
SSF *model*	*Single Stuck-Fault model*, üksiku konstantrikke mudel
state-of-the-art	arengu nüüdistase
TT	digitaalskeemide diagnostikatarkvara Turbo-Tester
XOR	*Exclusive Or*, välistav või

Sisukord

1 Sissejuhatus 9
1.1 Ülesande kirjeldus 10
1.2 Töö eesmärk 11
2 Digitaalsüsteemide diagnostika 12
2.1 BIST (*Built-In Self-Test*) 13
2.2 Funktsionaalne isetestimine 14
3 ALU operatsioonosa projekteerimine 16
3.1 Algoritm 16
3.2 Skeem 18
4 ALU juhtosa algoritmi modelleeriv programm 19
5 Analüüs 21
6 Kokkuvõte 23
Kasutatud kirjandus 24
Lisa 1 – Projekteeritud ALU 25
Lisa 2 – Projekteeritud ALU summaatori plokk 26
Lisa 3 – Programmi kood ALU.c 27
Lisa 4 – Programmi kood functions.h 31

Jooniste loetelu

Joonis 1. Baklaureusetöö kirjeldus [2] 10
Joonis 2. LFSR näide koos tekkivate pseudo-juhuslike vektoritega [2]..... 13
Joonis 3. Funktsionaalne isetestimine [5]..... 14
Joonis 4. Register..... 16
Joonis 5. Algoritm 17
Joonis 6. ALU skeem 18
Joonis 7. Käsurea info näide..... 19
Joonis 8. Faili „alu.tst“ näide..... 20

Tabelite loetelu

Tabel 1. TT ATPG tulemused 21
Tabel 2. Üksikute testide tulemused..... 21
Tabel 3. Ühiste testide tulemused..... 22

1 Sissejuhatus

Me elame ühiskonnas, kus igapäevane sõltuvus tehnoloogiast on väga suur. Iga päev kasutame me seadmeid, mis peidavad endas keerulisi tehissüsteeme, olgu siis selleks arvutid, mobiiltelefonid, autod, kuid ka suuremad süsteemid nagu näiteks lennukid või elektrijaamad. Kõikide nende ja paljude teiste argipäevaste seadmete tööd juhivad neis olevad mikrokiibid ja programmid. Vead nendes põhjustavad ootamatuid käitumisi kasutatavates seadmetes, mille tulemusena mõjutatakse süsteemiga seotud inimeste elusid vähemal või suuremal määral. Rike tehissüsteemi töös võib olla tühine – nt elektrooniline kraadiklaas ei näita õiget temperatuuri – või äärmiselt eluohtlik – nt lennuki autopiloodisüsteem lakkab töötamast.

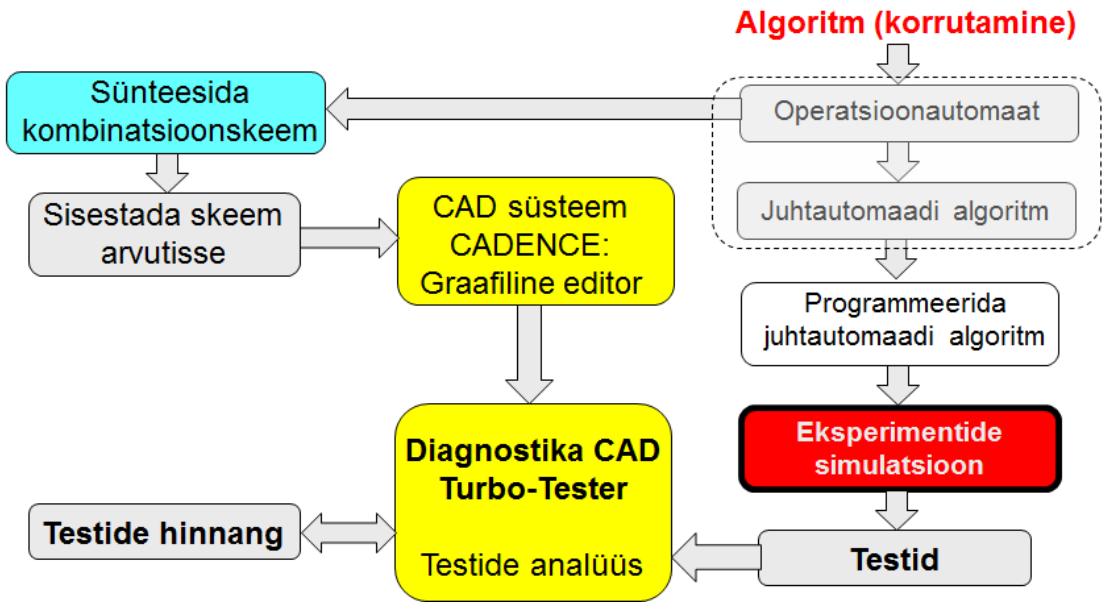
Aina suurenev tingitus tehnoloogiast on muutnud meid äärmiselt haavatavaks ja abituks olukordades, kus tehnika streikima hakkab. Seda olukorda märkame tavaliselt alles siis, kui on liiga hilja ning õnnetus on juba juhtunud. Seetõttu on tähtis süsteemide testimine ja diagnostika, et tagada veakindlus ja ära hoida potentsiaalseid õnnetusi tulevikus. Elektroonikasüsteemide tootmisel kulutatakse sageli kuni 70% kõikidest kuludest just testimisele. Kogu selle uue ning lõputult areneva tehismaailma keskel on üks vanasõna jätkuvalt tugevalt jõus – õnnetus ei hüüa tulles. Meie töö on end võimalikult hästi õnnetuste eest kaitsta ja neid vältida. [1]

Töö on jaotatud kolmeks põhiliseks peatükiks. Esimese peatükis tutvustatakse töö teoreetilist tausta – üldiselt digitaalsüsteemide diagnostikast ning isetestimisest. Teises peatükis antakse ülevaade kasutatud algoritmist ning projekteeritud ALU-st. Kolmandas peatükis seletatakse sisendjadade genereerimiseks loodud programmikoodi. Lõpus esitatakse tehtud töö kohta analüüs ning kokkuvõte. Lisades on välja toodud projekteeritud ALU ja ALU summaatori skeemid ning sisendjadade genereerimiseks loodud programmifailid „alu.c“ ja „functions.h“.

1.1 Ülesande kirjeldus

Antud bakalaureusetöö on aine „Digitaalsüsteemide diagnostika“ iseseisva kursusetöö laiendatud lahendus (Joonis 1). Töö kujutab endast funktsiooni valem (1) põhjal koostatud digitaalskeemi (ALU) funktsionaalse isetestimise võimekuse uurimist. Seade koosneb juht- ja operatsioonosast. Operandid a ja b on suvalised 8-bitised positiivsed kahendarvud.

$$y = (a+b)(a-b) \quad (1)$$



Joonis 1. Baklaureusetöö kirjeldus [2]

Kasutatavad tööriistad (eritarkvara) on digitaalskeemide projekteerimistarkvara CADENCE ja digitaalskeemide diagnostikatarkvara Turbo-Tester.

Lahendatavateks ülesanneteks on:

1. projekteerida ette antud funktsiooni täiteva ALU operatsioonosa ja selle sisestada see arvutisse CADENCE tarkvara abil;
2. koostada juhtosa algoritmi modelleeriv programm, mille ülesandeks oleks ALU operatsioonosa sisendsignaalide jada määramine seadme funktsiooni täitmise käigus;
3. analüüsida saadud ALU sisendjadade võimekust avastada ALU struktuurseid rikkeid, kasutades selleks Turbo-Testeri rikete simulaatorit.

Kasutatava algoritmi tsüklilise töö tulemusena tekkiv 8-bitine lõppvastus ei oma operandide a ja b suhtest sisulist tähendust, st ei ole funktsiooni valem (1) vastus.

1.2 Töö eesmärk

Esitatud uurimus on üheks tüüpiliseks etapiks digitaalsüsteemide projekteerimisel – testimismeetodi valimisel ning testitavuse kindlaks tegemisel ja hindamisel. Antud töö eesmärk on uurida projekteeritud kombinatsioonskeemi isetestimise võimekust ja välja selgitada need kitsaskohad, mis ei luba saavutada 100% rikete katet. Samuti teada saada, kas ja kui suurelt mõjutab operandide a ja b valik saavutatavat katteprotsenti ning saavutada parim võimalik tulemus antud algoritmi piirides.

2 Digitaalsüsteemide diagnostika

Digitaalsüsteemide diagnostika eesmärk laiemas mõttes on testimise abil kindlaks teha, kas süsteem töötab korralikult ning vigade korral lokaliseerida rikete asukoht ehk anda diagnoos. Digitaalsüsteemide testimiseks on vaja teste ehk testvektorite sisendjadasi, mis on eelnevalt genereeritud spetsiaalsete väliste testimisseadmete abil või süsteemi enda ressursside poolt. Testi kvaliteedi määrab protsent, mis iseloomustab avastatud rikete ja kogu süsteemi rikete vahelist suhet. Testimise juures on tähtis leida hinna ja kvaliteedi optimaalne suhe. Me võime testida väga põhjalikult ja saavutada ideaalis 100% kvaliteedi, kuid see nõuab palju aega ning on väga kulukas. Teiset küljest võime testimise peale investeerida minimaalselt aega ja raha, kuid seetõttu on süsteemi rikki minemise tõenäosus suurem, mis toob omakorda kaasa lisakulutusi trahvide ja muu näol. [1]

Rikete modelleerimisel kasutatakse loogilisi rikkeid, mis kujutavad skeemi füüsiliste rikete käitumist. Loogilised rikked võivad olla struktuursed, mis muudavad komponentide vahelisi ühendusi, või funktsionaalsed, mis muudavad komponentide funktsioone. Struktuursete rikete mudelid eeldavad, et loogikaelemendid on töökorras ning vead esinevad ainult neid ühendavates juhtmetes. Konstantrikked (s-a-0 ja s-a-1) ehk SSF mudel on standardne rikkemudel. Tema kasuks räägivad järgnevad omadused:

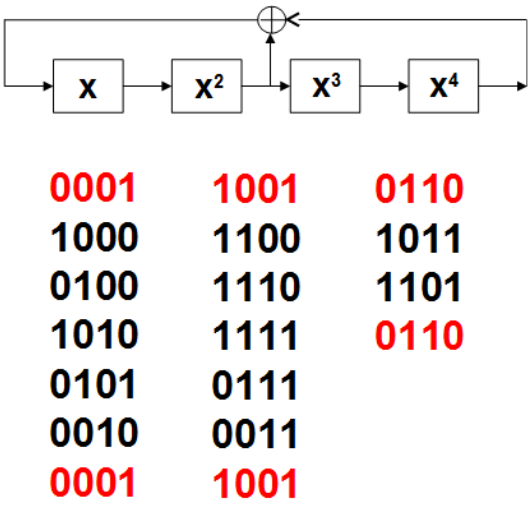
- esindab mitmeid erinevaid füüsilisi rikkeid;
- sõltumatus tehnoloogiast;
- testid, mis avastavad SSF rikkeid avastavad ka teisi mitmeid mitteklassikalisi rikkeid;
- võrreldes teiste mudelitega on SSF rikete arv skeemides väike;
- SSF rikkeid saab kasutada teiste rikete modelleerimiseks. [3]

Digitaalsüsteemide diagnostika juures on väga tähtis testimine. Skeeme saab testida väliste testerite abil, mis võivad olla väga kallid. Samuti on võimalik testida kasutades skeemi enda ressursse. Järgnevas kahes alapeatükis tutvustatakse lähemalt isetestimise teoreetilist tausta, millele tuginetes on lähenetud ka antud töös funktsionaalse isetestimise võimekuse hindamisele.

2.1 BIST (*Built-In Self-Test*)

Üks rakendatavaid testimise viise on sisseehitatud isetestimine (BIST), kus testskeemid asetatakse otse disainitavale tootele. See lubab neil paremini teste kontrollida ja jälgida, sest nad on lähemal testitavale funktsioonile Samuti saab testida seadet oma enda töökeskonnas normaalsel töötamiskiirusel (taktsagedusel). BISTi eelisteks on vähendatud sõltuvus välistest testseadmetest, mis võivad olla väga kallid. Siiski on vajalik spetsiaalse välise riistvara olemasolu testvektorite genereerimiseks ja testi väljundi hindamiseks. BIST aitab ka vähendada testandmete töötluseks vajaliku mälumahu suurust, kuna testimiseks võib vaja minna ainult algset sisendstiimulit ehk nn iduväärtust (*seed value*).

Sisseehitatud isetestimine (BIST) korral genereeritakse kombinatsioonskeemi sisendstiimulid pseudo-juhuslikult (PRG), mille tulemusena tekkivaid väljundeid analüüsitakse signatuuranalüsaatori (SA) poolt. Kui eesmärgiks on kõikide võimalike sisendkombinatsioonide läbitestimine, siis PRG võib olla lihtne n-bitine loendur. Kuid suuremate n väärtuste korral (n>19), osutub see ebapraktiliseks ja ebavajalikuks. Traditsioonilise BIST arhitektuuri korral kasutatakse sisendite genereerimiseks lineaarse tagasisidega nihkeregistrit (LFSR), mis tagab mõnesaja sisendvektoriga enamikel seadmetel piisava katteprotsendi. Joonisel (Joonis 2) on polünoomiga x^4+x^2+1 määratud lihtsa LFSR ehitus, mis iga järgmine takt nihutab vasakult sisse teise ja neljanda biti XOR tehte väärtuse. SA koondab sisendite tulemusena tekkinud kombinatsioonskeemi väljundid vektorisse, mida kutsutakse signatuuriks, mida võrreldakse oodatud signatuuriga, et teha kindlaks, kas loogikaskeem reageeris sisendstiimulile õigesti. [4]



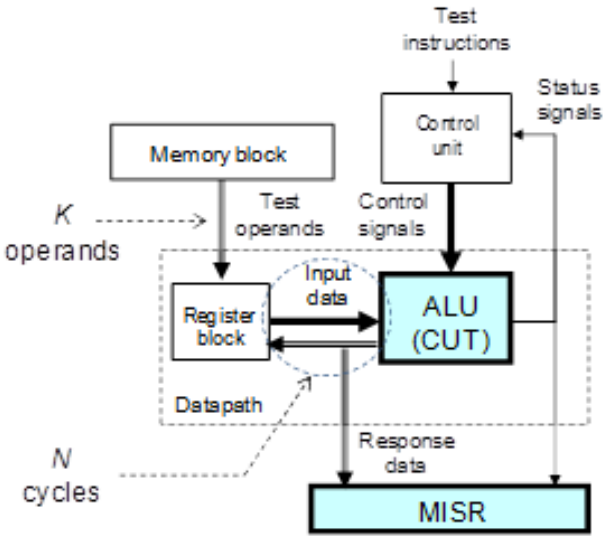
Joonis 2. LFSR näide koos tekkivate pseudo-juhuslike vektoritega [2]

2.2 Funktsionaalne isetestimine

Funktsionaalne BIST kujutab endas testimismeetodit, kus kontrollitakse skeemi funktsionaalseid mooduleid, et need genereeriks teste, mis tuvastaks struktuurseid rikkeid süsteemi teistes osades. Testvektorid toodetakse protsessori enda poolt kasutatdes süsteemis juba realiseeritud käske. Erinevalt *state-of-the-art* lähenemisest, ei ole vaja täiendavat välist spetsiaalset testimisriistvara ning puudub vajadus hoida testvektoreid skeemi mälus. Testvektoreid genereeritakse jooksvalt funktsionaalsete plokkide poolt süsteemile omasel normaalselt töökiirusel (taktsagedusel), mis tagab parema rikete katteprotsendi võrreldes traditsioonilise BIST lahendusega.

FBIST põhiidee seisneb skeemis juba olemasolevate funktsionaalsete protsesside kasutamises testide genereerimiseks ning testitava skeemi (CUT) väljundi monitoorimises mitme sisendiga signatuuranalüsaatori (MISR) poolt. MISR on ainuke lisariistvara, mis on vajalik FBIST implementeerimiseks. Protsessori tasemel, kasutatakse protsessori funktsionaalsust, et rakendada struktuurseid rikkeid igale CUT osale.

Joonis 3 kujutab funktsionaalse isetestimise näidet, kus testi all olevaks seadmeks on ALU (CUT).



Joonis 3. Funktsionaalne isetestimine [5]

Andmetee (*datapath*) koosneb registrite plokist (*register block*), kus ajutiselt salvestatakse ALU operatsioonides osalevad operandid. Näiteks jagamise operatsioonil salvestatakse registrites jagatav, jagaja, kõik vahetulemused ja tsükliloendur. CUT-i

sisenevaid registriplokkide andmeid (*input data*) ja juhtseadme (*control unit*) sisendsignaale (*control signals*) tõlgentatakse kui testvektoreid. ALU väljund saadetakse tagasi registrite plokki ning olekusignaalid (*status signals*) saadetakse tagasisidena juhtseadmesse. Kõike seda registreeritakse MISR poolt. Jagamise n tsükli tulemusena genereeritakse jooksvalt n testvektorit ning vastavalt tekib n signatuuranalüsaatori vastust, mis jälgib kogu ALU-s toimuvat jagamise protsessi. Terve mikrokäsutasemelise testimisprotsessi algatas jagamisekäsk kahe operandiga – jagatav ja jagaja. Erinevalt tuntud lähenemistest, kus käsud ja käsu tulemused on testi sisendid ja väljundid, on meil antud juhul olukord, kus iga tsükli operatsiooni kõik CUT-i sisendid loetakse ühtseks testvektoriks, mille väljundreaktsioone registreeritakse MISR poolt. Selle tulemusena on meil n korda rohkem testvektoreid, sest viisime testi ligipääsu käsu tasemelt mikrokäsu tasemele. [5]

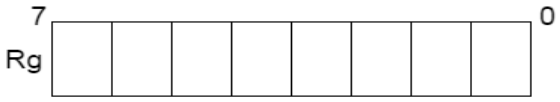
Funktsionaalse testimise vigade katteprotsenti parandamiseks ning seega ka testimise kvaliteeti tõstmiseks saab sama operatsiooni läbi viia mitmekordselt teiste operandidega. Probleemiks ongi leida optimaalseim operandide valik, mis annaks parima tulemuse ja lühendaks kogu testimise protseduuri.

3 ALU operatsioonosa projekteerimine

ALU operatsioonosa projekteerimiseks oli eelnevalt vaja selgeks määrata, millise algoritmi järgi tegutsetakse, et teada mitut registrit ja milliseid juhtsignaale vaja läheb. Kokku on kaheksa 8-bitist registrit ning algortimi iga operatsiooni tulemusena kirjutatakse ühe registri sisu üle. Järgnevalt seletatakse algoritmi ja ALU skeemi lähemalt.

3.1 Algoritm

Joonis 5 kujutab endas algoritmi vooskeemi. Alguses algväärtustatakse kõik 8 registrit vastavalt, samuti on võimaluse korral paremal näidatud alväärtustatud registri sisu kahendkoodis. Bittide järjestus läheb paremalt (noorimad bitid) vasakule (vanemad bitid), alustades nullist ning lõpetades seitsmega (Joonis 4).



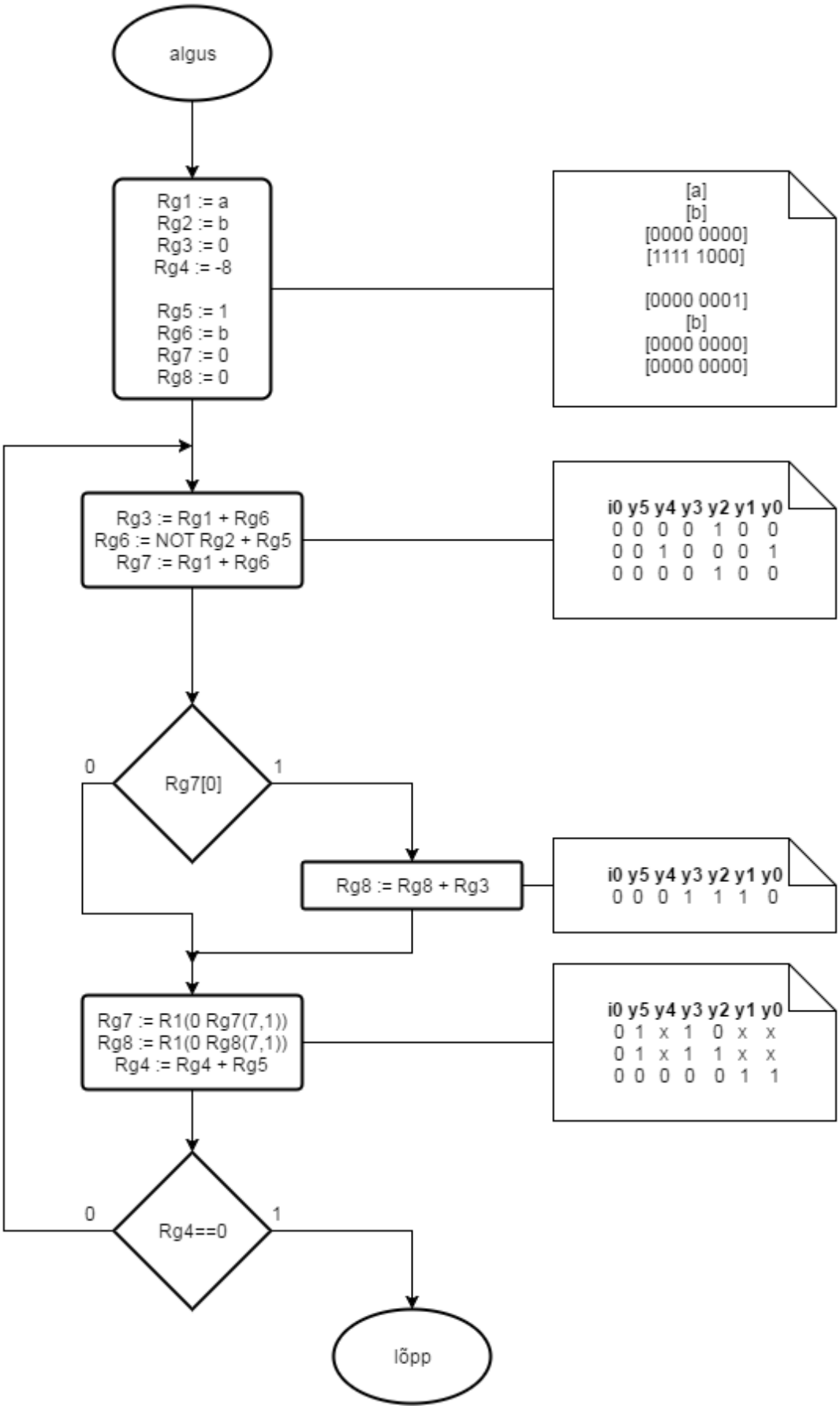
Joonis 4. Register

Peale algväärtustamist salvestatakse kolmandasse registerisse a+b väärtus ja kuuendasse registrisse b täiendkoodi väärtus ehk -b ning lõpuks seitsmendasse registrisse a-b väärtus.

Edasi minnakse tsüklisse, mida korratakse 8 korda. Esmalt kontrollitakse seitsmenda registri noorima biti (0-biti) väärtust. Kui see on 1, liidetakse eelnevalt saadud a+b ehk register kolme väärtus kaheksandasse registrisse, kuhu hakkab algoritmi tsüklilise töö tulemusena akumuleeruma korrutise vastus.

Järgmisena nihutatakse seitsmendat ja kaheksandat registrit ühe biti võrra paremale ning suurendatakse loendurit (Rg4) ühe võrra.

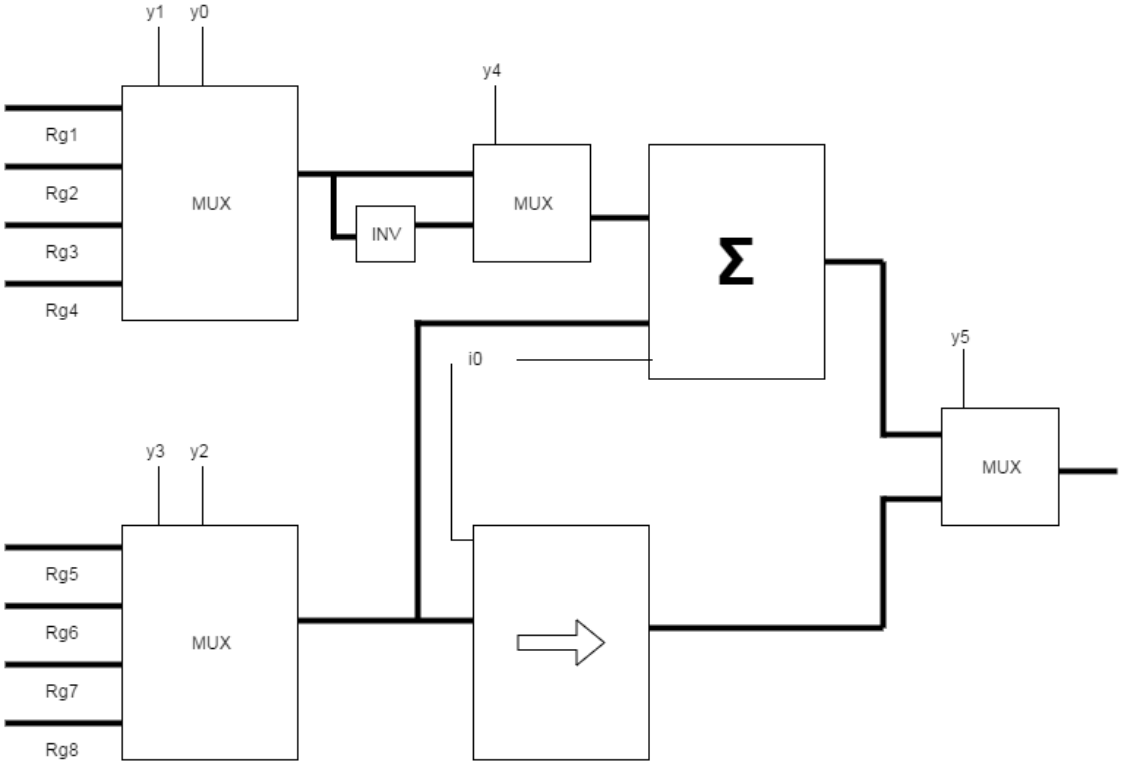
Lõpetuseks kontrollitakse, kas loendur on võrdne nulliga. Kui see tingimus ei ole täidetud korratakse tsüklit uuesti, vastasel korral jõutakse algoritmi lõppu.



Joonis 5. Algoritm

3.2 Skeem

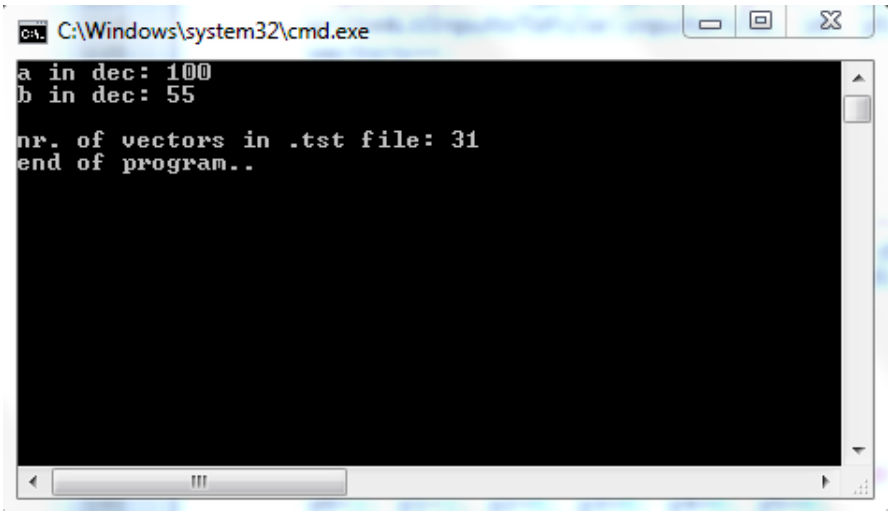
Projekteeritud ALU skeem on kujutatud järgmisel joonisel (Joonis 6). ALU sisendeid kontrollivad 2 multiplekserit (MUX). Esimene kontrollib registreid 1-4 juhtsignaalidega y0 ja y1. Teine kontrollib registreid 5-8 juhtsignaalidega y2 ja y3. Esimese multiplekseri väljund on summaatori ploki esimene sisendoperand. Juhtsignaal y4 määrab, kas operand on otse- või pöördkoodis. Teise multiplekseri väljund on summaatori ploki teine sisendoperand või nihutamise ploki sisend. Summaatori ja paremale nihutamise ploki väljundid on viimase multiplekseri sisenditeks, mis määrab juhtsignaaliga y5 kumb operatsiooni tulemus on väljundiks. Sisend i0 on pidevalt väärtusega 0 ning on summaatori noorimasse järgu sissetulev ülekanne ja paremale nihutamise tulemusena sissetulev vanim bitt. Juhtsignaalide väärtused algoritmi erinevate operatsioonide ajal on nähtavad algoritmis (Joonis 5). Väärtus „x“ tähendab, et selle operatsiooni ajal ei oma selle signaali väärtus tähtsust. Digitaalsüsteemide projekteerimistarkvarasse CADENCE sisestatud ALU ja eraldi tehtud summaator on nähtav töö lõpus lisades (LISA 1 ja LISA 2).



Joonis 6. ALU skeem

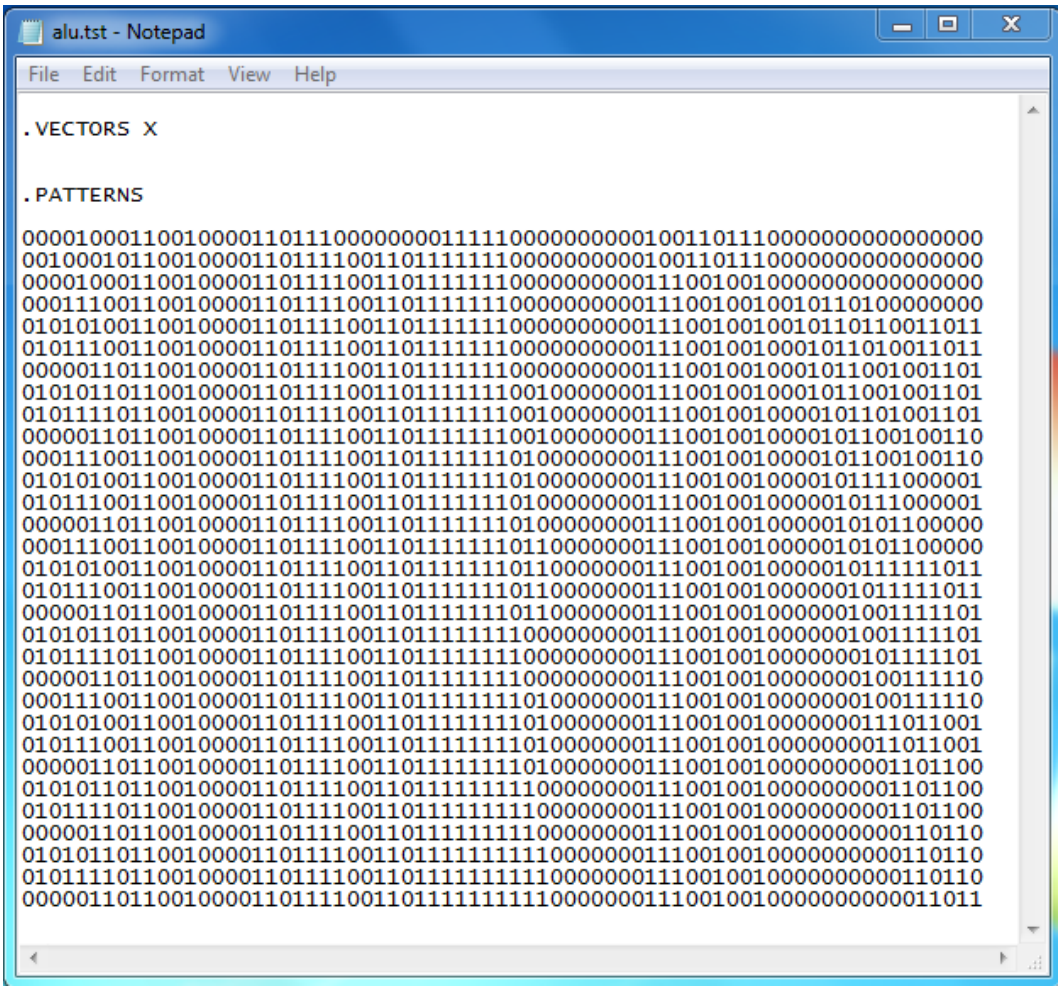
4 ALU juhtosa algoritmi modelleeriv programm

Programm koostati programmeerimiskeeles C kahes osas: failid „ALU.c“ (Lisa 3) ja „functions.h“ (Lisa 4). *Header* failis on olemas funktsioonid, mis imiteerivad projekteeritud alu erinevaid plokke (summaator, nihe paremale, inversioon), Samuti on seal funktsioonid kümnendsüsteemi arvu muutmiseks kahendkujule ja sisendvektorite kirjutamiseks vastavasse tekstifaili „alu.tst“. Sinna salvestatakse enne igat operatsiooni kõik ALU 71 sisendbitti ühe vektorina. Edukalt jooksutatud programm annab käsureal teada, mis on suvaliselt genereeritud operandide a ja b väärtused kümnendsüsteemis ja mitu sisendvektorit need operandid algoritmi läbimise tulemusena tekitasid (Joonis 7).



Joonis 7. Käsurea info näide

Tekkiva tekstfaili alguses on Turbo-Testeri jaoks vajalik formaat, kus on ära määratud, mitu vektorit on kokku (.VECTORS X). X tuleb siinkohal asendada käsurealt saadud vektorite arvuga. Peale järgmist märksõna (.PATTERNS) on esitatud kõik algoritmi tulemusena tekkinud vektorid järgmises järjestuses: i0, y5, y4, y3, y2, y1, y0, rg1[7-0], rg2[7-0], rg3[7-0], rg4[7-0], rg5[7-0], rg6[7-0], rg7[7-0], rg8[7-0] (Joonis 8).



Joonis 8. Faili „alu.tst“ näide

Selliseid testfaile saab kasutada koos Turbo-Testeriga, et analüüsida, kui hästi antud vektoritekogus avastab rikkeid skeemis, ja leida vigade katteprotsent.

5 Analüüs

Enne skeemi isetestimise võimekuse uurimist kasutati Turbo-Testeri automaatseid tööriistu (ATPG), mille abil veenduti, et skeemis ei ole tuvastamatuid rikkeid. Seda tõendab kõigi kolme meetodi (*deterministic, genetic, random*) poolt saavutatud 100% katteprotsent (Tabel 1).

Tabel 1. TT ATPG tulemused

	deterministic	***genetic***	***random***
Vektorite arv	62	27	53
Katteprotsent	100%	100%	100%

Projekteeritud ALU funktsionaalse iseteistimise võimekuse hindamiseks genereeriti loodud programmi abil 10 erinevat testfaili, mida sai analüüsida TURBO-TESTERI käsuga *ttanalyze*. Tulemused on nähtaval alloleval tabelis (Tabel 2).

Tabel 2. Üksikute testide tulemused

	a	**b**	**Vekorite arv**	**Kaetud rikked**	**Katteprotsent**
1.	100	55	31	486/694	70,028818
2.	48	217	32	440/694	63,400576
3.	105	128	32	423/694	60,951009
4.	236	240	33	442/694	63,688761
5.	156	46	32	474/694	68,299712
6.	24	165	32	472/694	68,011527
7.	164	190	32	477/694	68,731988
8.	49	36	30	441/694	63,544669
9.	132	174	32	471/694	67,867435
10.	39	251	30	448/694	64,553314
11.	0	0	27	289/694	41,642651
12.	255	255	27	351/694	50,576369
					62,608069

Tabelis on ära näidatud iga testi puhul operandide a ja b väärtused kümnendsüsteemis, nende operandidega läbikäidud algoritmi töö tulemusena tekkinud vektorite arv, kaetud rikete arv ning saavutatud katteprotsent. Viimasel real on ära toodud kõigi testide keskmine katteprotsent. Kahes viimases testis määrati a ja b väärtused nii, et üks kord oleks terve register täidetud nullidega (11. test) ja teine kord ainult ühtedega (12. test). Keskmise katteprotsendi järgi võib järeldada, et tulemused ei ole head. Parima, napilt üle 70%, katte saavutas esimene test.

Järgmiseks katsetati erinevate testide kombinatsioone, sest teadupärast funktsionaalse isetestimise juures kasutataksegi sama operatsiooni läbiviimiseks erinevaid operande, et parandada testimise kvaliteeti. Tabelis (Tabel 3) on ära näidatud kolme erineva testi tulemus. Esimeses ühises testis valiti üksikud testid 11 ja 12, sest see garanteeris, et iga operandidega seotud register on vähemalt korra olekus 0 ja 1, kuid see ei andud väga head tulemust. Teises ühises testis valiti üksikud testid 1 ja 7, sest need saavutasid omaette parimad katteprotsendid. Nende koostööl kasvas kate kergelt üle 9%.

Viimaseks pandi kõikide testide vektorid kokku, mille tulemusena saavutati 370 vektoriga 87,75% kate. Leian, et lisavektorite lisamine seda tulemust märgatavalt enam ei parandaks. See tuleneb algoritmi iseärasusest, mistõttu kolm registrit on pidevalt konstantsed – Rg 1, Rg 2 ja Rg 5.

Tabel 3. Ühiste testide tulemused

	Testide kombinatsioon	**Vektorite arv**	**Kaetud rikked**	**Katteprotsent**
1.	11. ja 12.	54	405/694	58,357349
2.	1. ja 7.	63	549/694	79,106628
3.	Kõik testid koos	370	609/694	87,752161

Arvestades, et kokku on digitaalskeemis 71 sisendit, millest 24 on pidevalt konstantsed, siis võib spekuleerida, et umbes pooled nende 24 biti poolt avastatavad rikked jäävad testimata. Ehk matemaatiliselt: 24/71=0,338; 0,338/2=0,169; 1-0,169=0,831. Tulemus 83% jääb lähedale saavutatud 87,8% kattele, mis lubab oletada, et saavutatud tulemus on reaalne ning kasutatud algoritmi piirides maksimumilähedane.

6 Kokkuvõte

Antud töö põhieesmärk oli digitaalskeemi funktsionaalse isetestimise võimekuse hindamine ja halvasti testitavate kohtade välja selgitamine. Töös kasutatud funktsionaalse sisseehitatud isetestimise lähenemisega saavutati katteprotsent 87,75%. Arvestatdes kasutatud algoritmi iseärasusi, mille tulemusena osa digitaalskeemi sisendeid on koguaeg konstantsed, võib väita, et saavutatud tulemus on maksimumilähedane. Operandide a ja b valik algoritmi ühekordse töö tulemusena mõjutas saavutatud katteprotsenti 30% ulatuses. Üksikute testide keskmine tulemus jäi 63% juurde.

Isetestimise võimekust saaks tõsta kasutades paremat algoritmi, mis peaks silmas, et ei oleks konstantseid sisendeid, kuid suure tõenäosusega ei tagaks see 100% katet. Loomulikult algoritmi modifitseerimise tulemusena muutuks ka ALU skeem, mis võib veel lisaks mõjutada saavutatavat katteprotsenti.

Kuna eesmärk on siiski 100% kate, siis selle saavutamiseks oleks kaks võimalust:

1. fikseerida valitud testiga mittetestitavad kohad, mille testimiseks valida täiendavalt (või ka valitud testi välja vahetamiseks) vastavalt mingi muu süsteemi funktsioon, kui see on võimalik (so tarkvaraline ehk „pehme“ lähenemine);
2. parandada skeemi testitavust aparatuursete täienduste abil, milleks võiksid olla täiendavad väljundid (jälgitavuse parandamiseks) või täiendavad sisendid (juhitavuse parandamiseks), mille abil õnnestuks skeemis tsükliliselt ringlevaid signaale vajalikul moel modifitseerida (juhtida).

Veel oleks üheks võimaluseks kombineerida traditsioonilist BIST-i funktsionaalse isetestimisega, kus BIST-i kulutused oleksid siiski väiksemad kui tema täiskujul realiseerimisel.

Kasutatud kirjandus

[1] Ubar, R. Digitaalsüsteemide diagnostika I: Diagnostiline modelleerimine. Tallinn : TTÜ Kirjastus, 2005

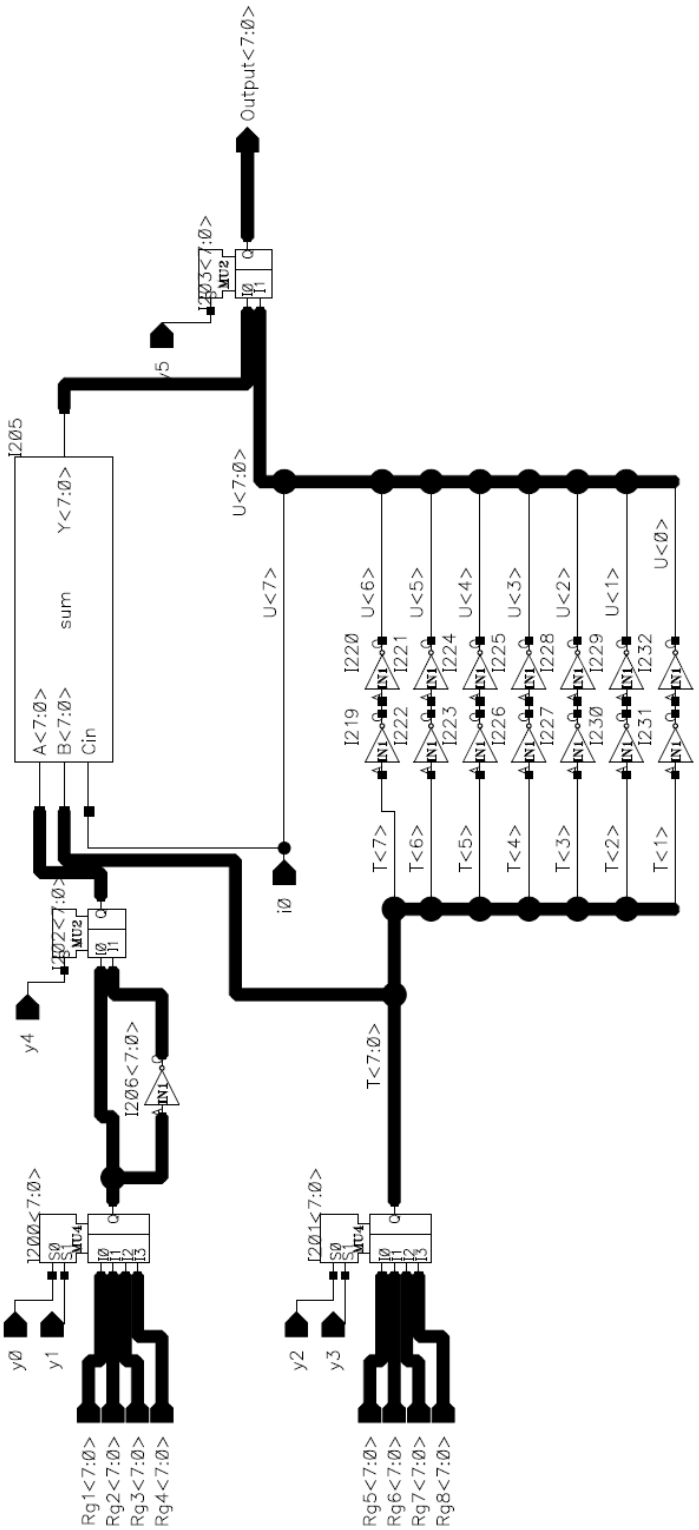
[2] [WWW] www.pld.ttu.ee/~raiub/web_0103/diagnostika/loengukiled/ (11.05.2016)

[3] [WWW] www.pld.ttu.ee/testing/theory/ (13.05.2016)

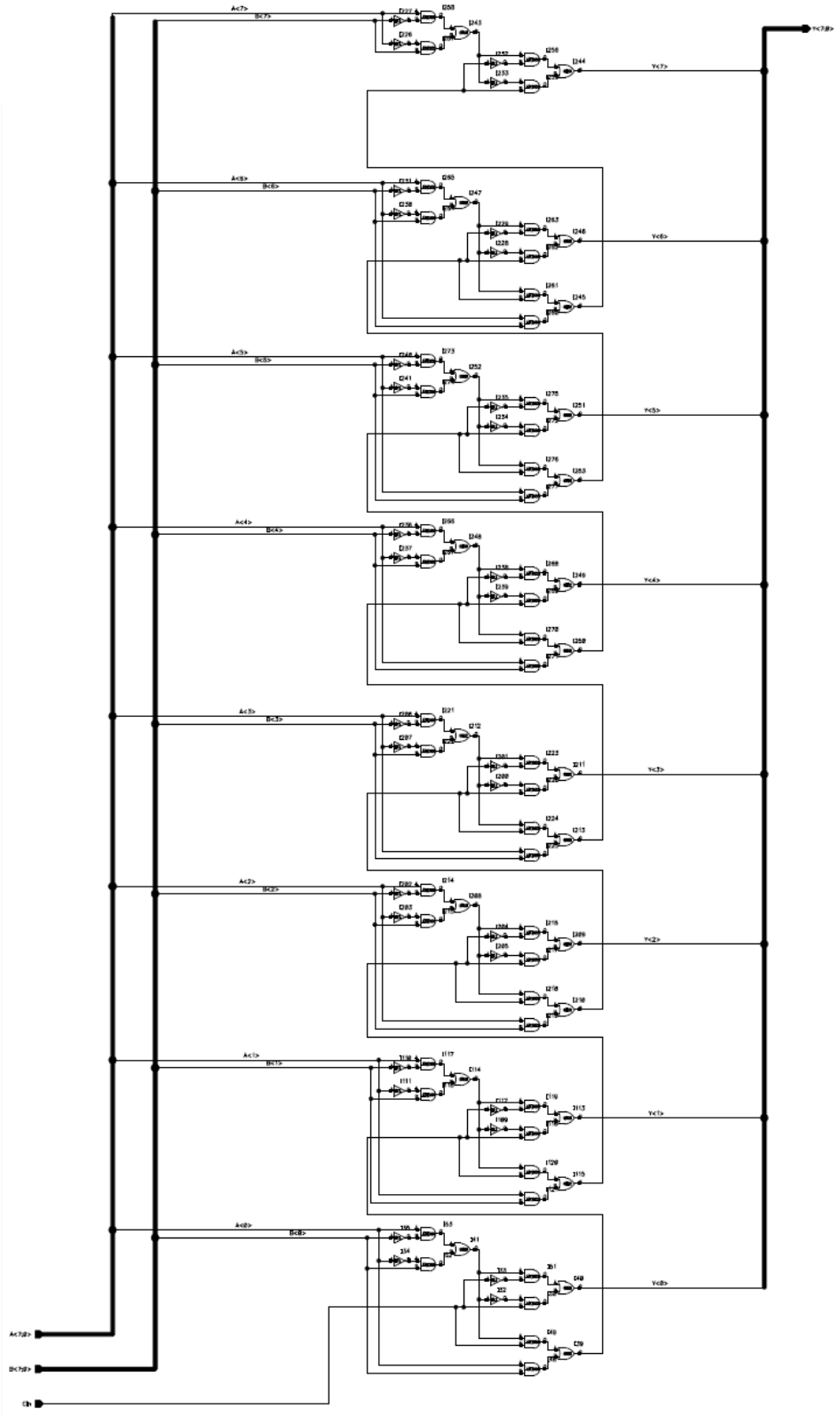
[4] Miczo, A. Digital Logic Testing and Simulation: Second Edition. USA : John Wiley & Sons, 2003

[5] Raimund Ubar, Viljar Indus, Oliver Kalmend. At-Speed Functional Built-In Self-Test Methodology for Processors. IASTED International Conference on Engineering And Applied Science –EAS 2012. Colombo, December 27-29, 2012

Lisa 1 – Projekteeritud ALU



Lisa 2 – Projekteeritud ALU summaatori plokk



Lisa 3 – Programmi kood ALU.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "functions.h"  // header file that contains all the
needed functions

int main(void){
    // file, where to save the input vectors (8*8+7=71 bits
total)
    char inputsFile[]="alu.tst";

    // correctly format the .tst file
    FILE *fp;       // file pointer
    fp=fopen(inputsFile, "a");
    if(fp!=NULL){
        fprintf(fp, "\n.VECTORS X\n\n\n.PATTERNS\n\n");     // X
- number of vectors
    }
    else{
        printf("file opening error in ALU.c");
    }
    fclose(fp);

    // generate random numbers a and b for equation y=(a+b)*(a-
b)
    // between 0 and 255 (unsigned 8-bit min-max)
    srand(time(NULL));
    int a, b, n=256;
    a = rand()%n;
    b = rand()%n;
    printf("a in dec: %d\n", a);
    printf("b in dec: %d\n", b);

    int *p;                             // pointer to first
element of returned array
    int i;                              // cycle counter
    int vectors=0;                      // number of vectors

    int y0, y1, y2, y3, y4, y5;         // mux control signals
    int i0=0;                           // constant 0 for ALU's
right shift and sum block

    // initialisation of 8-bit registers rg1-rg8
                                        // start value
explanation
    int rg1[8];                         // a
const
```

```
    p=decNrToBinRg(a);
    for(i=0; i<8; i++){
        rg1[i]=*(p+i);
    }

    int rg2[8];                         // b
const
    p=decNrToBinRg(b);
    for(i=0; i<8; i++){
        rg2[i]=*(p+i);
    }

    int rg3[8]={0,0,0,0,0,0,0,0};       // 0
a+b
    int rg4[8]={1,1,1,1,1,0,0,0};       // -8 (tAiendkoodis)
counter
    int rg5[8]={0,0,0,0,0,0,0,1};       // 1
const

    int rg6[8];                         // b
-b
    p=decNrToBinRg(b);
    for(i=0; i<8; i++){
        rg6[i]=*(p+i);
    }

    int rg7[8]={0,0,0,0,0,0,0,0};       // 0
a-b
    int rg8[8]={0,0,0,0,0,0,0,0};       // 0
cumulative sum

    // calculate a+b --- rg3:=rg1+rg6
    y0=0; y1=0; y2=1; y3=0; y4=0; y5=0;
    writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1, y0,
rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
    vectors++;
    p=sum(rg1, rg6);
    for(i=0; i<8; i++){
        rg3[i]=*(p+i);
    }

    // calculate NOT(b)+1 aka -b --- rg6:=rg2+rg5
    y0=1; y1=0; y2=0; y3=0; y4=1; y5=0;
    writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1, y0,
rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
    vectors++;
    p=notRg(rg2);
    for(i=0; i<8; i++){
        rg2[i]=*(p+i);
    }
    p=sum(rg2, rg5);
    for(i=0; i<8; i++){
        rg6[i]=*(p+i);
    }
```

```
    // change rg2 back to old value, because in ALU it actually
is just the inversion of the rg
    p=decNrToBinRg(b);
    for(i=0; i<8; i++){
        rg2[i]=*(p+i);
    }

    // calculate a+(-b) --- rg7:=rg1+rg6
    y0=0; y1=0; y2=1; y3=0; y4=0; y5=0;
    writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1, y0,
rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
    vectors++;
    p=sum(rg1, rg6);
    for(i=0; i<8; i++){
        rg7[i]=*(p+i);
    }

    // adding (a+b) in cycle to cumulative sum
    do{
        if(rg7[7]==1){
            // calculate sum+(a+b) --- rg8:=rg8+rg3
            y0=0; y1=1; y2=1; y3=1; y4=0; y5=0;
            writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2,
y1, y0, rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
            vectors++;
            p=sum(rg8, rg3);
            for(i=0; i<8; i++){
                rg8[i]=*(p+i);
            }
        }
        // shift (a-b) right by one bit --- rg7->
        /*y0=x; y1=x;*/y2=0; y3=1;/* y4=x;*/ y5=1;
        writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1,
y0, rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
        vectors++;
        p=shiftR(rg7);
        for(i=0; i<8; i++){
            rg7[i]=*(p+i);
        }

        // shift sum right by one bit --- rg8->
        /*y0=x; y1=x; */y2=1; y3=1;/* y4=x;*/ y5=1;
        writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1,
y0, rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
        vectors++;
        p=shiftR(rg8);
        for(i=0; i<8; i++){
            rg8[i]=*(p+i);
        }

        // calculate counter+1 --- rg4:=rg4+rg5
        y0=1; y1=1; y2=0; y3=0; y4=0; y5=0;
```

```
        writeALUInputsToFile(inputsFile, i0, y5, y4, y3, y2, y1,
y0, rg1, rg2, rg3, rg4, rg5, rg6, rg7, rg8);
        vectors++;
        p=sum(rg4, rg5);
        for(i=0; i<8; i++){
            rg4[i]=*(p+i);
        }
    }while(checkIfRgIsZero(rg4)==1);

    printf("\nnr. of vectors in .tst file: %d\n", vectors);
    printf("end of program..");
    getchar();
    return 0;
}
```

Lisa 4 – Programmi kood functions.h

```
#include <stdio.h>
#ifndef _FUNCTIONS_
#define _FUNCTIONS_

// declaration of function prototypes
int *decNrToBinRg(int);          // decimal number to binary
register
int *notRg(int[]);               // inversion block of ALU
int *sum(int[], int[]);          // sum block of ALU
int *shiftR(int[]);              // right shift block of ALU
int checkIfRgIsZero(int[]);      // check if counter rg is zero
void writeALUInputsToFile(char[], int, int, int, int, int, int,
int, int[], int[], int[], int[], int[], int[], int[], int[]);

// functions
int *decNrToBinRg(int decNr){
    static int binRg[8];
    int i;
    for(i=7; i>=0; i--){
        binRg[i]=decNr%2;    // mod 2 determines i bit value
        decNr=decNr/2;       // divide by 2 for the next bit
    }
    return binRg;
}

int *notRg(int notRgInput[]){
    static int notRgOutput[8];
    int i;
    for(i=7; i>=0; i--){
        if(notRgInput[i]==1){
            notRgOutput[i]=0;    // 1 --> 0
        }
        else{
            notRgOutput[i]=1;    // 0 --> 1
        }
    }
    return notRgOutput;
}

int *sum(int A[], int B[]){
    static int sumOutput[8];
    int cIn=0;
    int i;
    for(i=7; i>=0; i--){
        sumOutput[i]=A[i]+B[i]+cIn;
```

```
        if(sumOutput[i]>1){
            sumOutput[i]=sumOutput[i]%2;    // mod 2 determines
sum's i bit value
            cIn=1;                          // either way, cIn
into the next bit is 1
        }
        else{
            cIn=0;                          // else cIn into the
next bit is 0
        }
    }
    return sumOutput;
}

int *shiftR(int shiftRInput[]){
    static int shiftROutput[8];
    int i;
    for(i=7; i>=0; i--){
        if(i==0){
            shiftROutput[i]=0;                  // 0 bit moves
in from the left
        }
        else{
            shiftROutput[i]=shiftRInput[i-1];   // all others
shift to the right
        }
    }
    return shiftROutput;
}

int checkIfRgIsZero(int rg[]){
    int value=0;
    int i=0;
    for(i=7; i>=0; i--){
        if(rg[i]==1){
            value=1;            // not all bits in rg are zero
        }
    }
    return value;
}

void writeALUInputsToFile(char fileName[], int i0, int y5, int
y4, int y3, int y2, int y1, int y0, int r1[], int r2[], int
r3[], int r4[], int r5[], int r6[], int r7[], int r8[]){
    FILE *fp;       // file pointer
    fp=fopen(fileName, "a");
    if(fp!=NULL){
        // append new line of vectors to .tst file
        fprintf(fp,
"%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%
d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%d%
d%d%d%d%d%d%d%d\n",
                    i0, y5, y4, y3, y2, y1, y0,
```

```
                    r1[0],r1[1], r1[2], r1[3], r1[4],r1[5],
r1[6], r1[7],
                    r2[0],r2[1], r2[2], r2[3], r2[4],r2[5],
r2[6], r2[7],
                    r3[0],r3[1], r3[2], r3[3], r3[4],r3[5],
r3[6], r3[7],
                    r4[0],r4[1], r4[2], r4[3], r4[4],r4[5],
r4[6], r4[7],
                    r5[0],r5[1], r5[2], r5[3], r5[4],r5[5],
r5[6], r5[7],
                    r6[0],r6[1], r6[2], r6[3], r6[4],r6[5],
r6[6], r6[7],
                    r7[0],r7[1], r7[2], r7[3], r7[4],r7[5],
r7[6], r7[7],
                    r8[0],r8[1], r8[2], r8[3], r8[4],r8[5],
r8[6], r8[7]);
    }
    else{
        printf("file opening error in functions.h");
    }
    fclose(fp);
}
#endif
```