TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

Martin Perman

PSEUDOAMMENDAVATE TESTIDE GENEREERIMINE MIKROPROTSESSORI ARITMEETIKA-LOOGIKASEADMELE

Bakalaureusetöö

Juhendaja: Raimund-Johannes Ubar
Professor

Konsultant: Stephen Oyeniran Adeboye

Tallinn 2017

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: Martin Perman

13.05.2017

Annotatsioon

Töö käsitleb pseudo-ammendavate testide genereerimist erineva ehitusega summaatoritele, aritmeetika-loogikaplokile ja korrutamisseadmele ning võrdleb seda deterministliku automaatselt genereeritud testidega. Pseudo-ammendavate testide genereerimiseks pakutakse välja ka uut meetodit. Läbiva ülekandega summaatoritele on pseudo-ammendava testi genereerimine teistel põhimõtetel lihtsustatud tänu summaatori iteratiivsele ehitusele. Keerulisemates summaatorites seda teha ei saa ja lisaks pseudo-ammendavale testile tuleb testvektoreid juurde kaotades pseudo-ammendava testi eelised. Õnnestub aga võita testi pikkuses kui mõlemad testid kokku tõsta ja optimeerida. ALU testide tegemine toimub samal pseudojuhuslikul põhimõttel, kusjuures vaatluse alla ei ole võetud seadme juhtloogika testimine. Korrutajale rakendatakse uut lineaarse keerukusega testimismeetodit, mis küll praegustes katsetes ei andnud parimat tulemust, aga ka korrutaja tegelik skeem ei olnud teada.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 21 leheküljel, 6 peatükki, 8 joonist, 15 tabelit.

// # Abstract

Pseudoexhaustive test generation for ALU circuits of microprocessors

The thesis contains pseudoexhausteve test generation for different types of adders, ALU circuit and suggests new method for pseudoexhaustive test generation. It appears that it is quite easy to generate pseudoexhaustive test for ripple-carry adder circuit of any length, but it is not the case with fast-carry adders. It is possible to analyze fast adders with the same pseudoexhaustive test, but it will not cover all possible faults in the circuit. To cover all the faults, it is needed to generate extra testvectors for that circuit. It is also possible to combine both manually generated pseudoexhaustive test and ATPG tests, where after optimizing the resulted test has the properties of both tests combined: full SAF coverage, and shorter test. ALU circuit testing was done in a similar way to adder testing. It must be noted however, that ALU has complex driver logic, that cannot be tested in the same manners, so some of ALU faults were unable to be tested. ALU funcions had exhaustive tests generated for them, and were seperately tested as well as together. Separate testing shows how much do ALU functions share the same hardware and how much each test will add to the total fault coverage. The new pseudoexhaustive test generation method was described thoroughly and as example used on a multiplier circuits from miniMIPS processor. It must be noted again that circuit inner structure was not fully known and thus the generated test simulation did not result in full SAF coverage, but it still covered almost all of the possible SAF. This method should be under a further investigation.

The thesis is in estonian and contains 21 pages of text, 16 chapters, 8 figures, 15 tables.

Lühendite ja mõistete sõnastik

ALU	*Arithmetic logic unit,* aritmeetika-loogikaplokk
PA	*Pseudo-exhaustive*, pseudo-ammendav
D	deterministlik
ATPG	*Automated Test Pattern Generation,* automaatne testide genereerimine
SAF	*Stuck at fault*, konstant tüüpi rike

Sisukord

1 Sissejuhatus....................9
2 Süsteemide diagnostikast....................11
2.1 Ülevaade testide genereerimise meetoditest digitaalskeemidel....................11
2.2 Pseudo-ammendava testimise eesmärk ja põhimõte....................12
3 Summaatorite testimine....................14
3.1 Pseudo-ammendavad testid....................15
3.2 Deterministlik testide genereerimine....................17
3.3 Kombineeritud genereerimine....................17
4 ALU....................20
4.1 Uurimise eesmärgid ja eksperimendi keskkond....................20
4.2 ALU operatsioonide pseudo-ammendavad testid....................21
4.3 ALU kumulatiivne pseudo-ammendav test....................22
5 Korrutaja testimine....................24
5.1 Korrutamisseadme pseudo-ammendava testimise idee....................25
5.2 Eksperimendid MiniMIPS kombinatoorse korrutamisseadmega....................28
6 Kokkuvõte....................32
Kasutatud kirjandus....................33
Lisa 1 – 4-bitiste summaatorite testide genereerimine....................34
Lisa 2 – 8-bitiste summaatorite testide genereerimine....................35
Lisa 3 – 16-bitiste summaatorite testide genereerimine....................36
Lisa 4 – ALU testide genereerimine....................37
Lisa 5 – ALU testide genereerimine jätk....................38
Lisa 6 – Korrutaja testide genereerimine....................39
Lisa 7 – Korrutaja A-operandi vektorid....................41
Lisa 8 – Korrutaja B-operandi vektorid....................42

Jooniste loetelu

Joonis 1. Võimalik rikete arv erinevates summaatorites..15
Joonis 2. Katmata jäänud rikete protsent erinevates summaatorites...............................16
Joonis 3. Turbo-Testeriga genereeritud testide arv erinevate summaatorite jaoks..........17
Joonis 4. Deterministliku ja pseudo-ammendava testide kokkupaneku vektorite protsendi vähenemine..18
Joonis 5. ALU funktsioonide testide katteprotsendid eraldi testimisel...........................22
Joonis 6. ALU rikete katteprotsendi kasv funktsioonide testide lisamisega...................23
Joonis 7. 4-bitiste arvude korrutamine...24
Joonis 8. Lihtne korrutaja skeem..25

Tabelite loetelu

Tabel 1. Järjestikülekandega summaatori pseudo-ammendava testi genereerimine.......15
Tabel 2. Lihtsustatud ALU käsustik..........20
Tabel 3. Korrutaja pseudo-ammendava testi genereerimine..........28
Tabel 4. Testvektorid Korrutaja A-sisendile..........29
Tabel 5. Testvektorid korrutaja B-sisenditele..........29
Tabel 6. Eksperimendid 4 korrutajaga..........30
Tabel 7. 4-bitiste summaatorite testvektorid..........34
Tabel 8. 8-bitiste summaatorite testvektorid..........35
Tabel 9. 16-bitiste summaatorite testvektorid..........36
Tabel 10. ALU funktsioonide testvektorid..........37
Tabel 11. ALU funktsioonide testvektorid jätk..........38
Tabel 12. Korrutaja testide genereerimine..........39
Tabel 13. Korrutaja A-operandi vektorid..........41
Tabel 14. Korrutaja B-operandi vektorid..........42

1 Sissejuhatus

Digitaalsed süsteemid on juba väga levinud ja nende töökindlusest sõltub palju. Maailm üha rohkem sõltub süsteemide usaldatavusest. Enne seadme kasutuselevõttu peab üle kontrollima, kas seade täidab oma funktsiooni õigesti, kuna digitaalsete skeemide tootmine on väga täpne protsess ja rikked on kerged tulema.

Teades süsteemi ehitust, on võimalik talle anda erinevaid sisendeid ja oodata õiget väljundit. Kui väljund on vale, siis on teada, et mingi komponent, mida etteantud sisendid juhivad, on vigane.

Rikked võivad olla erinevat tüüpi: konstant tüüpi rikked, kus signaali juhe on lühises kõrge või madala signaaliga (*stuck-at faults*), või lühis/tühis erinevate komponentide vahel, mis muudab süsteemi funktsiooni. Kui rikkeid on rohkem kui üks, siis võib esineda ka rikete maskeerimine.

Süsteemi täielikuks testimiseks olekski vaja genereerida kõikvõimalikud sisendite kombinatsioonid. See võib tegelikkuses võtta liiga palju aega, sest süsteemi sisendeid võib olla palju ja kombinatsioonide arv kasvab eksponentsiaalselt. Mäluga elementide testimine on veelgi keerulisem, kuna ei saa piirduda ainult sisendite kombinatsioonidega, vaid peab arvestama ka eelmisi süsteemi olekuid.

Näiteks on välja arvutatud, et Intel 8080 protsessori täielikult ammendav test võtaks aega 37 aastat, aga tootja tegi testi mis võttis 10 sekundit. Põhjuseks see, et osasid funktsioone ei kasutata süsteemi eluea vältel.[2]

Töös on kasutatud rikete testimiseks Turbo-Testeri tarkvara [7] ja Cadence skeemiredaktorit, mis on kättesaadav TTÜ Infotehnoloogia Maja arvutiklassides.

Töö peamiseks eesmärgiks oli uurida pseudo-ammendavate testide genereerimise metoodikat digitaalskeemide testimise eesmärgil ja uurida selle võimalusi ning kvaliteeti erinevate skeemitüüpide puhul, mida kasutatakse arvutite ja mirkoprotsessorite ALU-des.

Alguses tutvustatakse süsteemi diagnostikat. Sellele järgnevad 4-, 8- ja 16 järguliste, läbiva ja kiire ülekandega summaatorite pseudo-ammendavate (PA) testide pikkuste ja

kvaliteeti võrdlemised deterministlikke (D) testidega. Uuritakse ka ALU rikete katteprotsenti erinevate ALU operatsioonide pseudo-ammendavate testide puhul. Lõpus uuritakse ka korrutaja testide genereerimise probleemi.

2 Süsteemide diagnostikast

Süsteemide diagnostikast rääkides, peab paika panema ka diagnostika põhimõtted ja rakendatavad meetodid.

Testvektor on sisendite kombinatsioon mis annavad konkreetse väljundi ja vastavalt signaalide liikumise järgi väljundisse on võimalik tuvastada signaali teekonnale jäävad vead, mis võisid tekkida skeemi füüsilisel realisatsioonil. Näiteks skeemi valmistamisel võib tekkida lühis lähestikku olevate skeemi elementide vahel, ning selle tõttu skeem ei anna enam õiget tulemust ega realiseeri esialgset funktsiooni. Vahest võib vigu tekkida ka korraga mitu tükki, mis teatud olukordades võivad maskeerida teineteist. Selles töös ei uurita maskeerimise probleemi.

Töö katseskeeme diagnostikas viidi läbi programmiga Turbo-Tester, mis analüüsib digitaalskeemi võimalikke vigu ja on võimeline riketele vastavaid teste genereerima kasutades automaatseid testide genereerimise algoritme [7] .

2.1 Ülevaade testide genereerimise meetoditest digitaalskeemidel

Süsteemide diagnoosimist saab jagada kahte põhilisse lähenemisviisi. Need on funktsionaalne testimine ja struktuurne testimine.

Funktsionaalne test kontrollib, kas skeem käitub vastavalt tema kirjeldusele. Funktsionaaltest on täielik, kui on testitud kõik sisendite kombinatsioonid, ehk rakendatud ammendav test (*exhaustive test*).

Võib ka vaadata kõikide rikkega funktsioonide hulka, kui testobjekti. Nüüd rakendades skeemile testvektoreid, saab hakata rikutud funktsioone välistama vastaselt skeemi väljunditele. Seega esimese testiga juba saab välistada 50% riketest. Teise testiga 75%, kolmandaga 87,5%, neljandaga 93,75% jne. 100%-ni jõudmiseks on aga siiski vaja kõik sisendite kombinatsioonid läbi proovida ja tihti selline testimisviis läheb liiga pikaks, et seda rakendada.

Struktuurse testimise puhul vaadatakse skeemi struktuuri poolt määratud rikkeid. Kuna kõikvõimalikke rikkeid pole võimalik alati leida, siis lepitakse ainult ühte tüüpi rikete otsimises. Tavaliselt on need konstant tüüpi rikked (*SAF*). Struktuursed testid ei anna tavaliselt kohe suuri rikete katteid, vaid iga test annab igakord katteprotsendile juurde mingi osa kogu riketest. Seega see meetod on testi alguses aeglasem kui funktsionaalne testimine, aga selle täielik rikete koondumine ei aeglustu nagu funktsionaalse meetodi puhul.

Testi kvaliteeti saab hinnata mitme parameetriga: kui palju vigu katab test kogu rikete hulgast, kui palju test avastab tõestatud liiaseid rikkeid (liiased rikked ei muuda skeemi funktsiooni), kui pikk on test ja kui palju aega võtab.

Testide genereerimist võib vaadata kui riket avastava testvektori otsimist kõikide kombinatsioonide hulgast. Kombinatsioonide arv aga kasvab eksponentsiaalselt süsteemi sisendite arvuga, seega genereerimine peab olema effektiivne.

Üks võimalik meetod on juhuslike arvude meetod. Sellega genereeritakse süsteemile juhuslikke sisendeid ja kontrollitakse millised rikkeid genereeritud vektor katab. See meetod aga ei suuda garanteerida kõrget rikete katet ja pole võimalik tõestada rikete liiasust. Raske on ka siis, kui skeemis esineb vigu, mida on võimalik avastada ainult ühe sisendite kombinatsiooniga (raskesti avastatavad rikked).

Teine võimalus testvektoreid genereerida on deterministlik meetod, mis keskendub vektorite otsimisele ühele konkreetsele veale. Protsessi käigus saavad ka osad teised vead testituks. Meetodeid saab ka kombineerida omavahel, kasutades ära juhusliku meetodi kiirust alguses ja deterministliku meetodi täpsust pärast.

2.2 Pseudo-ammendava testimise eesmärk ja põhimõte

Ammendav (*exhaustive*) test tähendaks kõikide võimalike sisendite kombinatsioonide läbiproovimist. Nii saavad kõikvõimalikud skeemi rikked alati avastatuks. Teisest küljest muutub selline test väga kiiresti tülikaks, sest paljude sisenditega skeemides on liiga palju võimalikke sisendite kombinatsioone, et neid jõuaks kõiki ära kontrollida mõistliku aja jooksul ja selle testi puhul ei tohi olla skeemis mäluelemente ega mäluefektidega vigu, sest siis muutub skeem olekutest sõltuvaks. Kõik testid ei avasta ka alati uusi vigu ja nende kasutamine muutub mõtetuks.

Pseudo-ammendav test kujutab endast väikseimat testvektorite kogumit, mis korraga paralleelselt kontrollib skeemide sees olevaid alamskeeme ja simuleerimisel annab tulemuseks kõikide võimalike rikete katte. See on võimalik tänu sellele, et mitme väljundiga skeemidel reeglina üks väljund ei sõltu kõikidest sisenditest vaid ainult osadest. Näiteks lihtsa järjestikülekandega summaatoril kõikide ühebitiste lihtsummaatorite testimist saab viia läbi paralleelselt.

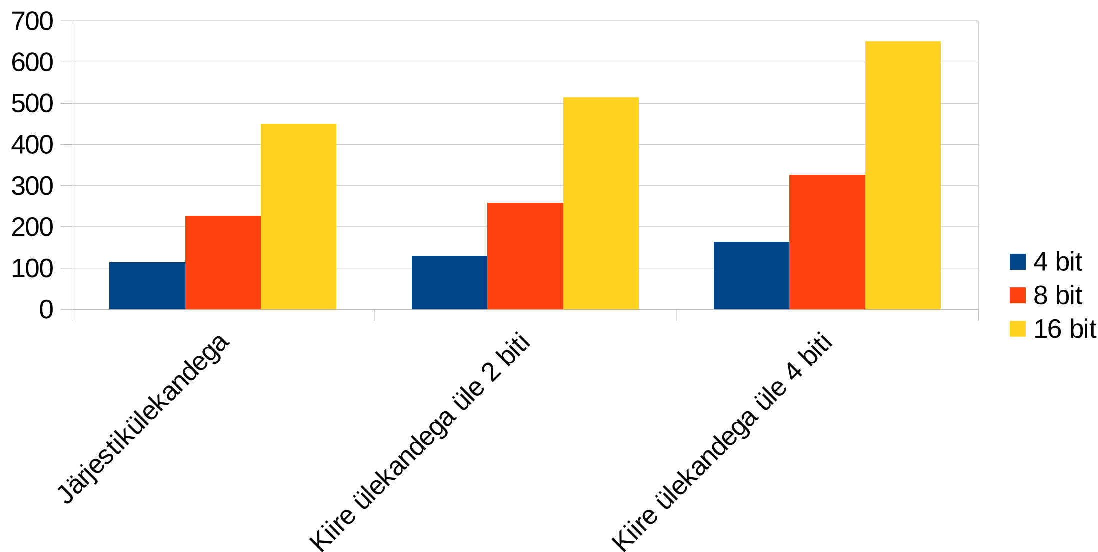
Pseudo-ammendav testimine ei kata ainult SAF vead, vaid ka kõik teised vead avastatakse, kaasaarvatud ka kordsed vead, sest meetod sisuliselt rakendab kõikidele alamskeemidele ammendavad testid. Samas on testi pikkus märgatavalt lühem kui see oleks kogu skeemile rakendatud ammendava testi puhul.

3 Summaatorite testimine

Käesoleva peatüki eesmärgiks on koostada regulaarse struktuuriga järjestikülekandega täisarvude summaatorile pseudoammendav test ja hinnata selle kvaliteeti SAF-tüüpi rikete (SAF – *Stuck-at-Faults*, konstant 1 ja 0 tüüpi rikked) suhtes. Seejärel vaadeldakse meetodi kasutatavust summaatorite puhul, kus regulaarsust on „rikutud“ nt. kiire ülekande realiseerimiseks sisse viidud skeemide poolt (regulaarseks nimetame skeemi, kus erinevad järgud on täpselt sama struktuuriga).

Summaatorite võrdlemiseks on eraldi koostatud kolme erineva järkude arvuga ja kolme erineva ülekandega summaatorid. Kokku 9 erinevat summaatorit. Ülekandetüüpideks oli valitud: järjestikülekanne, kiire ülekanne üle 2 biti ja kiire ülekanne üle 4 biti ning summaatorite järgupikkusteks oli valitud 4, 8 ja 16 järku. Kõik summaatorid said koostadud Cadence skeemiredaktoriga ja konverteeritud Turbo-Testeris kasutatavateks .agm failideks.

Kiire ülekande üle 8 biti tegemine nõuab juba märgatavalt rohkem riistvara [5] , seega antud töös piirduti 4 bitise kiire ülekandega. Graafikult (Joonis 1) on ka näha kuidas rikete hulk summaatoris kasvab summaatori keerukuse kasvamisega. Seega seda keerulisem on ka testide automaatne genereerimine ja testide suure hulga tõttu on ka hilisem rikete kontroll vastavalt pikem.



Joonis 1. Võimalik rikete arv erinevates summaatorites

3.1 Pseudo-ammendavad testid

Summaatori pseudo-ammendav test kujutab endast testverktorite kogumit, mis korraga paralleelselt rakendavad kõikidele lihtsummaatoritele ammendavad testid. Ehk kõikides lihtsumaatorite sisendites käivad läbi kõik kombinatsioonid. Kuna lihtsummaator liidab vaid kahte bitti ja ülekannet, siis tema erinevaid sisendkombinatsioone on kokku kaheksa. Tuleb välja, et kaheksa vektoriga saab testida ükskõik kui suure järkude arvuga summaatorit [6] .

Tabel 1. Järjestikülekandega summaatori pseudo-ammendava testi genereerimine

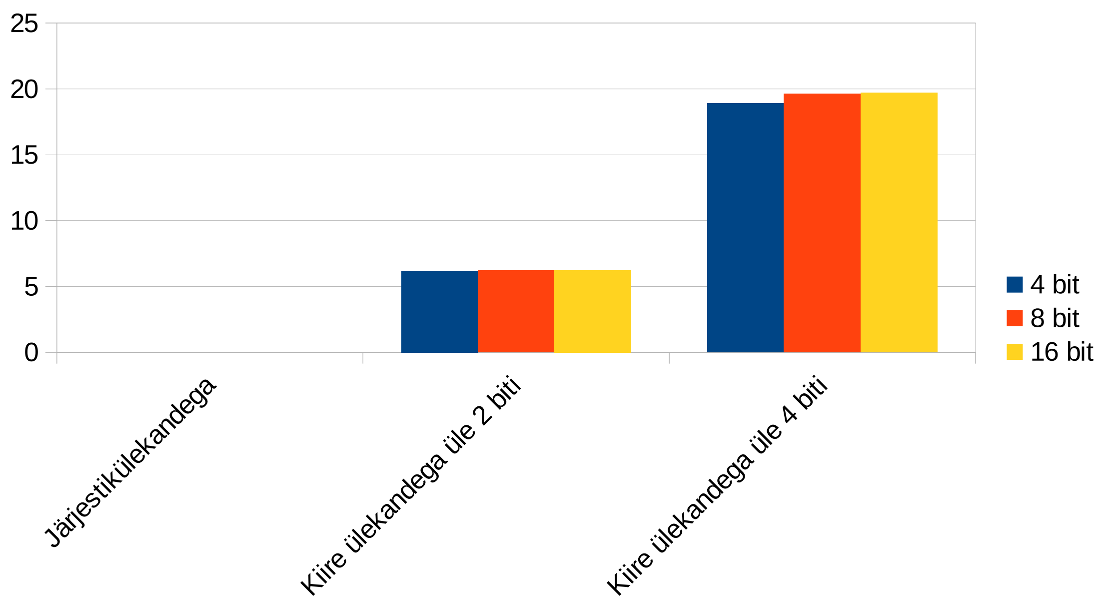
No	…	4-bit	3-bit	2-bit	1-bit	0-bit
		a_4 b_4 c_4	a_3 b_3 c_3	a_2 b_2 c_2	a_1 b_1 c_1	a_0 b_0 c_0
1	…	0 0 0	0 0 0	0 0 0	0 0 0	0 0 0
2	…	0 1 0	0 1 0	0 1 0	0 1 0	0 0 1
3	…	1 0 0	1 0 0	1 0 0	1 0 0	0 1 0
4	…	1 1 0	0 0 1	1 1 0	0 0 1	0 1 1
5	…	0 0 1	1 1 0	0 0 1	1 1 0	1 0 0
6	…	0 1 1	0 1 1	0 1 1	0 1 1	1 0 1
7	…	1 0 1	1 0 1	1 0 1	1 0 1	1 1 0
8	…	1 1 1	1 1 1	1 1 1	1 1 1	1 1 1

Tabelis (Tabel 1) on toodud regulaarse järjestikülekandega summaatori test. Kõige vasakpoolsemas alamtabelis on kõik summaatori esimese järgu sisendmuutujad a_0 b_0 c_0

vabad ja ammendava testi koostamine (kõik 8 erinevat koodi) on lihtne. Järgmise järgu jaoks on aga tekkinud nüüd kitsendus ülekandemuutuja c1 väärtuste näol, mis on määratud 0-järgu sisendmuutujatega. Kuid ka selles järgus õnnestub juba ette antud c_1 väärtuste jaoks kõik 8 koodi tabelisse sobitada. Teisese biti puhul toimime samamoodi kui esimese biti puhul. Alates kolmandast bitist aga selgub, et kahe külgneva alamtabeli testid hakkavad korduma ja *copy-paste* meetodil on nüüd suvalise pikkusega järjestikülekande summaatorile pseudo-ammendavat testi lihtne genereerida.

Kaheksast vektorist koosnevat pseudo-ammendavat testi saab rakendada kahjuks ainult järjestikülekandega summaatorile. Samade vektorite rakendamine kiire ülekandega summaatorites enam ei anna täieliku rikete katte, sest kiire ülekandega summaatorites on ülekannete paralleelseks arvutamiseks eraldi funktsioonid, mis ei ole enam nii regulaarsed nagu järjestikülekandega summaatoril. Mida kaugemale arvutatakse kiiret ülekannet, seda rohkem kulub selleks riistvara ja seda rohkem vigu on vaja skeemis kontrollida.

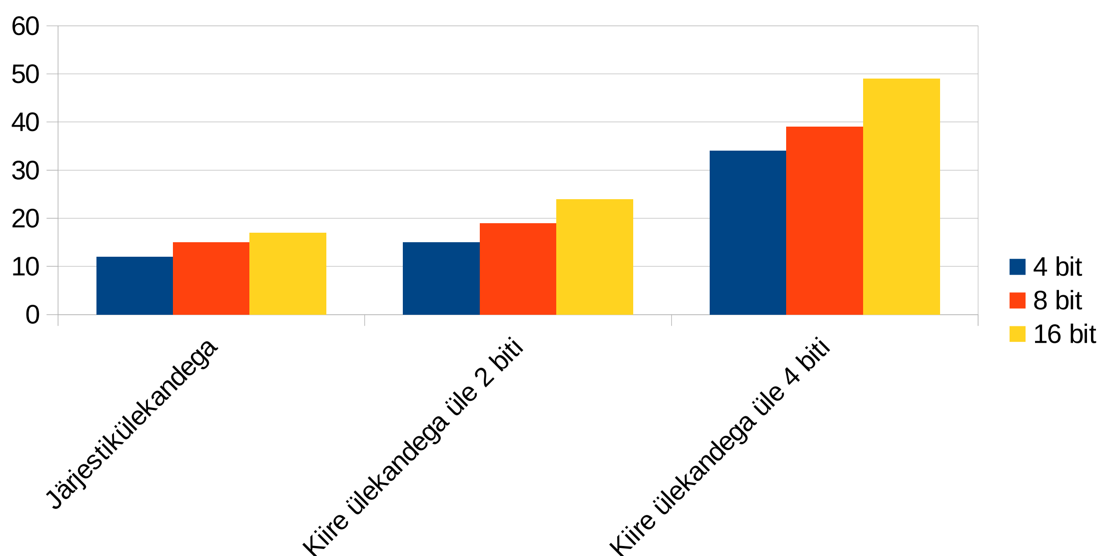
Üks katsetest oligi summaatori pseudo-ammendava testi katsetamine kiire ülekandega summaatorite peal. Tulemust on näha järgnevalt graafikult (Joonis 2). Nii nagu võikski oodata, mida suurema kiire ülekandega summaator on, seda rohkem jääb testimata vigu. Muidugi järjestikülekandega summaatoris kaetakse ära kõik vead.



Joonis 2. Katmata jäänud rikete protsent erinevates summaatorites

3.2 Deterministlik testide genereerimine

Deterministlik automaatne testide genereerimine (ATPG) on Turbo-Testeri funktsioon, kus programm ise genereerib antud skeemile erinevaid teste, et katta ära võimalikult palju skeemi vigu. See algoritm toimib hästi väiksemate skeemide puhul ja praegustes katsetes andis tulemuseks suhteliselt väiksed testvektorite hulgad kõikide rikete kattmiseks. Suuremate skeemide puhul täielik rikete kate automaatse testide generaatoriga võib võtta liiga palju aega ja tulemuseks ei kaeta ära skeemi kõik võimalikud vead. Graafikult (Joonis 3) on võimalik näha kuidas automaatselt genereeritud testide arv kasvab sumaatori keerukuse kasvamisega. Siin võib ka kõrvale märkida ka seda, et käsitsi koostades pseudo-ammendavaid teste järjestikülekandega summaatoritele peaks alati andma kaheksa vektorit sõltumata summaatori järguarvust. Automaatne testide genereerimine andis aga summaatori järkuarvu kasvamisega igakord pikema testi. Seega juba on näha, et automaatne testide genereerimine ei anna alati parimat tulemust.



Joonis 3. Turbo-Testeriga genereeritud testide arv erinevate summaatorite jaoks

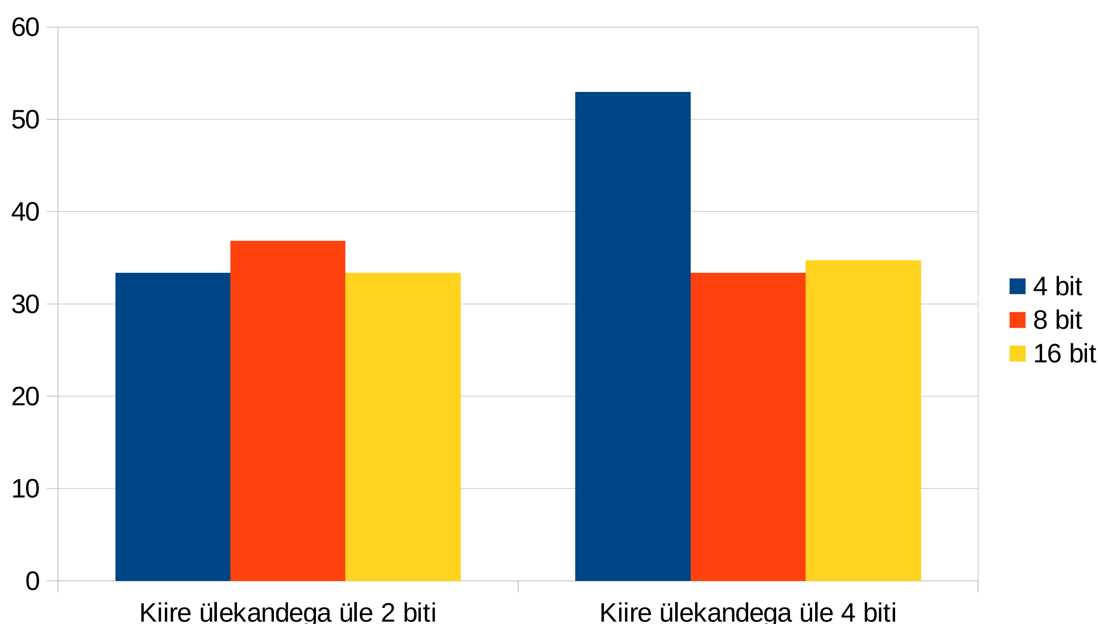
3.3 Kombineeritud genereerimine

Kuna esialgne pseudo-ammendav test ei kata kõiki võimalikke vigu kiire ülekandega summaatorites, aga katab valdava enamuse ja on lühike ning deterministlik automaatne testide genereerimine annab palju vektoreid, siis üks läbiviidud katse oli mõlemat

meetodit kokku viia ja vaadata, kas tulemuseks on ikka täielik rikete kate, kuid väiksema vektorite arvuga.

Sellist operatsiooni sai teostada Turbo-Testeri funktsiooniga *optimize*. Funktsioon vaatab üle kõik vektorid ja eemaldab mittevajalikud ning optimeerib olemasolevaid. Iga summaatori jaoks oli vaja mõlema eelneva meetodi genereeritud vektorid käsitsi kokku tõsta. Seejärel lasta käima *optimize* funktsioon.

Tulemuseks on õnnestunud saavutada stabiilselt kõikidel summaatoritel vektorite hulga vähenemise võrreldes automaatselt genereeritud versiooniga. Vektorite arv kahanes keskmiselt 34% võrra võrreldes automaatselt genereeritud vektorite arvuga. Seda on näha ka graafikus (number).



Joonis 4. Deterministliku ja pseudo-ammendava testide kokkupaneku vektorite protsendi vähenemine

Esines ka üllatavalt palju parem tulemus neljabitise kiire ülekandega summaatori puhul, kus vektorite protsent kahanes 53% võrra. Selline tulemus võib tuleneda sellest, et sellel summaatoril ei ole enam sisemisi ülekandeid. Kui teistel kiire ülekandega summaatoritel oli ülekanne üle kahe või nelja biti välja arvutatud, siis kuna see summaator on kõigest neljabitine ja tema ülekanne arvutatakse peale neljandata bitti, läheb see kohe välja kui genereeritud ülekanne.

Selle meetodi puuduseks on see, et kombineerimise ja optimeerimise käigus kaotatakse tegelikult pseudo-ammendava testi head omadused, kuna optimeerimine keskendub

ikkagi SAF rikete leidmisele. Positiivsest küljest test katab kõik SAF tüüpi vead ja testi pikkus on lühem kui ainult deterministlikutl genereeritud test, seega sobib olukordades, kus ei õnnestu pseudo-ammendava testiga katta kõik vead, nagu kiire ülekandega summaatorite näites.

4 ALU

Arithmetic logic unit (ALU) on aritmeetika-loogikaseade, mis teostab erinevaid operatsioone andud operandidega. See on tähtis protsessori osa, ilma milleta ei oleks võimalik andmeid töödelda. Selle testimise probleemi võiks ära lahendada vastavalt igale funktsioonile pseudo ammendava testi genereerimisega.

4.1 Uurimise eesmärgid ja eksperimendi keskkond

Peatüki eesmärgiks on uurida psudo-ammendava testimise metoodika rakendamist mikroprotsessori ALU-le rervikuda. Uuritavaks protsessoriks on valitud instituudis hetkel uurimistöös kasutatav MiniMIPS, mille kohta on instituudis olemas uurimiseks vajalik protsessori mudel ja ka simuleerimise tööriistad. Vaatluse all olev ALU on töö tulemuste selguse mõttes lihtsustatud.

Tabel 2. Lihtsustatud ALU käsustik

ADD	Operandide summa märgi mittearvestamisega
ADDU	Operandide summa märgi arvestamisega
SUB	Operandide vahe märgi mittearvestamisega
SUBU	Operandide vahe märgi arvestamisega
AND	Bitthaaval loogiline ja tehe operandide vahel
OR	Bitthaaval loogiline või tehe operandide vahel
XOR	Bitthaaval loogiline välistav või tehe operandide vahel
NOR	Bitthaaval loogiline või-ei tehe operandide vahel
SLL	Loogiline nihe vasakule
SRL	Loogiline nihe paremale
SRA	Aritmeetiline nihe paremale
LUI	*Load upper immediate*

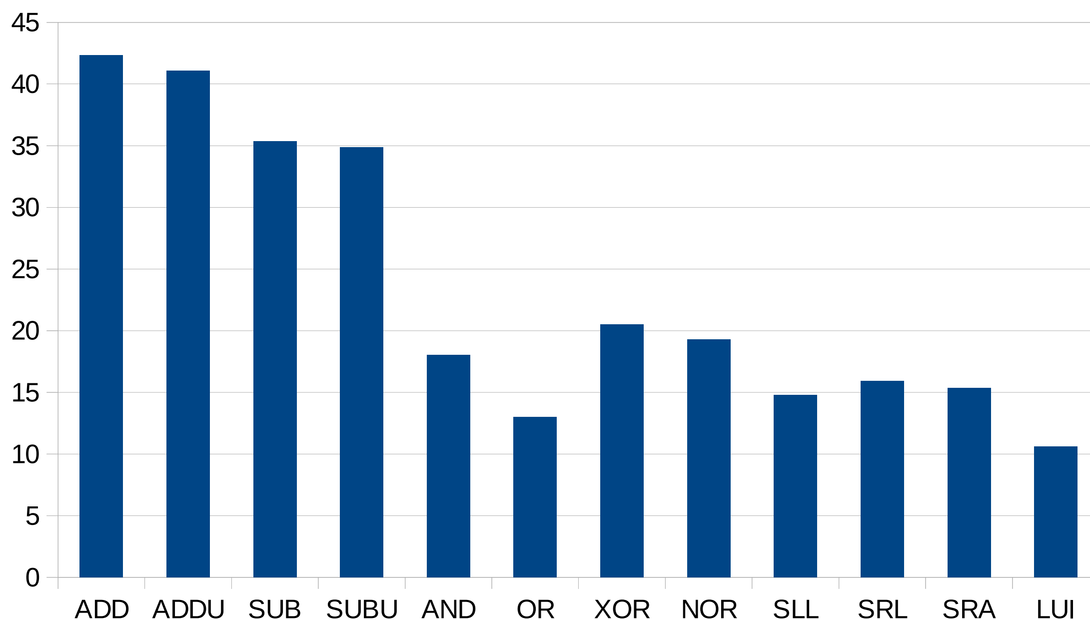
4.2 ALU operatsioonide pseudo-ammendavad testid

Antud ALU funktsioone saab jagada kolme gruppi: 3-, 2- ja 1-bitised operatsioonid. 3-bitised, ehk aritmeetika opearatsioonid on summeerimised ja lahutamised (ADD, ADDU, SUB, SUBU), sest nende teostamiseks kasutatakse kolme bitti: a_i, b_i ja c_i. Kus a_i ja b_i on lihtsummaatori operadid ja c_i ülekanne. 2-bitised operatsioonid on loogikaoperatsioonid: AND, OR, NOR ja XOR. 1-bitised operatsioonid on nihke- ja lihtedastuskäsud: SLL, SRL, SRA ja LUI.

Katses vaadeldi ALU igat operatsiooni eraldi ning uuriti kuidas iga operatsioonile eraldi genereeritud pseudo-ammendav test katab ära skeemis võimalikud vead. Teoorias kui alus on kõik funktsioonid ehitatud paralleelselt, siis neid eraldi testides ja katteprotsendid kokku liites peaks saama natuke üle 100%, sest iga test testib ka alu funktsioone ühendavat riistvara.

Katse annab ka ettekujutuse kui keerukad või mahukad on erinevad operatsioonid ALU skeemis. Mida keerulisemad funktsioonid on, seda rohkem riistvara nad hõlmavad ja seega nende testimisel peaks olema kaetud rikete arv suurem. Graafikult (Joonis 5) on näha kuidas erinevate funktsioonide testimine katab erinevaid vigu kogu ALU skeemist. Ühtlasi on näha, kuidas summaator on valdavalt suurim antud ALU funktsioon. Testvektorite genereerimisel kasutati käsitletud pseudo-ammendavate testide genereerimise meetodeid. Summaatorite testid on juba olemas, teised operatsioonide testid said nende näitel samamoodi pseudo-ammendavad testid, kasutades ära skeemide paraleelselt testitavust kõikides operatsioonides. Testvektorid on lisatud töö lisadesse (Tabel 10 ja Tabel 11).

Kõikide funktsioonide rikete katteprotsendid kokkuliites saab tulemuseks 281%, mis tähendab, et funktsioonid ei ole realiseeritud paraleelselt üksteisega, vaid ALU skeemi on optimeeritud funktsioone kokkutõstes ja seega antud ALU funktsioonid omavad palju ühiseid osi omavahel. Nende testid aga eraldi simuleerimisel katavad palju ühiseid ALU skeemiosi. Protsent 281% iseloomustab ilmekalt kõigi katsetatud käskude alla kuuluvate skeemiosade ülekattuvuste kogumahtu.



Joonis 5. ALU funktsioonide testide katteprotsendid eraldi testimisel

On näha, kuidas aritmeetika operatsioonide testimine katab tunduvalt suurema osa ALU riketest, kui teised operatsioonid. Loogika operatsioonide testimine katab keskmiselt natuke rohkem vigu, kui nihke- ja lihtedastusoperatsioonid, ühe erandiga OR operatsiooni suhtes, kus katteprotsent on märgatavalt väiksem.

4.3 ALU kumulatiivne pseudo-ammendav test

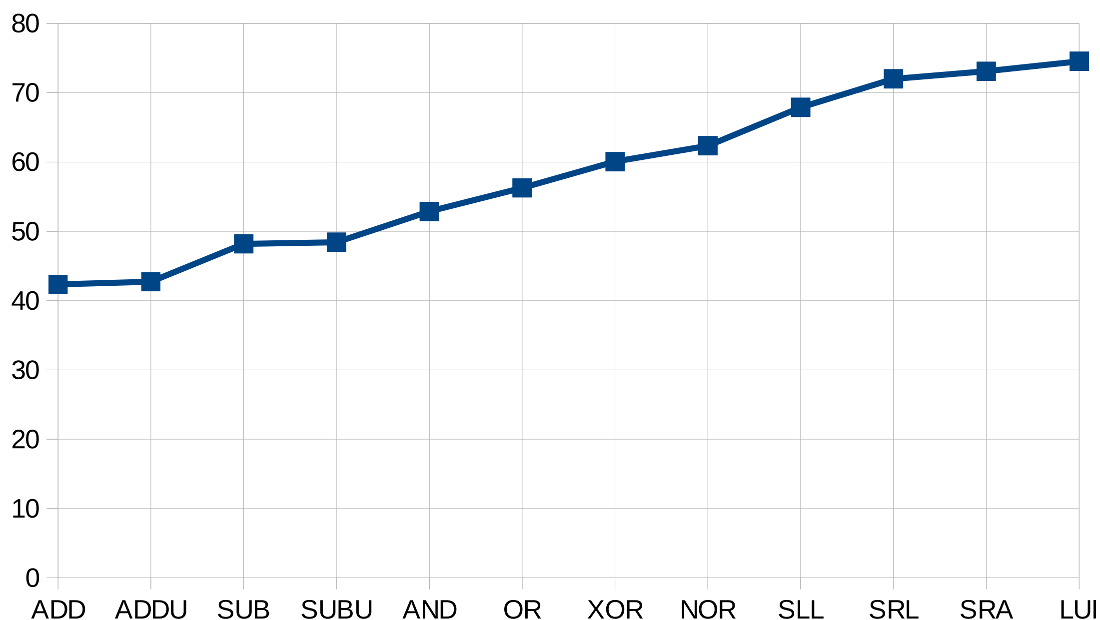
Katse nägi ette alu testvektorite hulga pidevat täiendamist iga funktsiooni pseudo-ammendavate testidega ja uurida käskude „panuste" kumulatiivset kasvu (rikete katte suurenemist). Seega oleks võimalik ka näha, kuidas mõned ALU funktsioonid on omavahel seotud tihedamalt ja kasutavad ühesuguseid osi rohkem kui teised.

Andud juhul sai alustatud aritmeetikakäskudest just seetõttu, et saaks võrrelda eraldi nii aritmeetika klassi käskude omavahelist „panuse" võrdlust, kui ka sama ülejäänud käskude osas.

Saadud graafik on enamasti ühtlase tõusuga, mis tähendab, et kõikidel käskudel on ikkagi oma kindel (ja rikete katte suhtes oluline) roll olemas, v.a. märgi ja märgi liitmis ning lahutamisoperatsioonide puhul. Graafik oleks loomulikult teistsugune kui funktsioonide järjestus oleks teine.

Tulemusi on näha graafikul (Joonis 6). Näiteks on selgelt näha, et summaator on antud alu üks kaalukamaid osi, sest tema testid annavad juba peaaegu poole ALU rikete

kattest. Samas on näha ka, et lahutamise funktsioon väga palju kattub tavalise liitmisega, mis oleks ka loogiline, sest lahutamine on sisuliselt negatiivse arvu juurde liitmine.



Joonis 6. ALU rikete katteprotsendi kasv funktsioonide testide lisamisega

Küll aga alu kogu struktuur ei olnud katsete tegemise ajal teada ja seega kõikide rikete katet ei õnnestunud saada. Katmata jäänud vead jäävad ALU juhtloogika osaks, mida käsitsi ei oska testida. ALU oli ka märgatavalt lihtsustatud katsete läbiviimiseks ja peale lihtsustamist võis kustutatud funktsioonide juhtloogika ikka alles jääda.

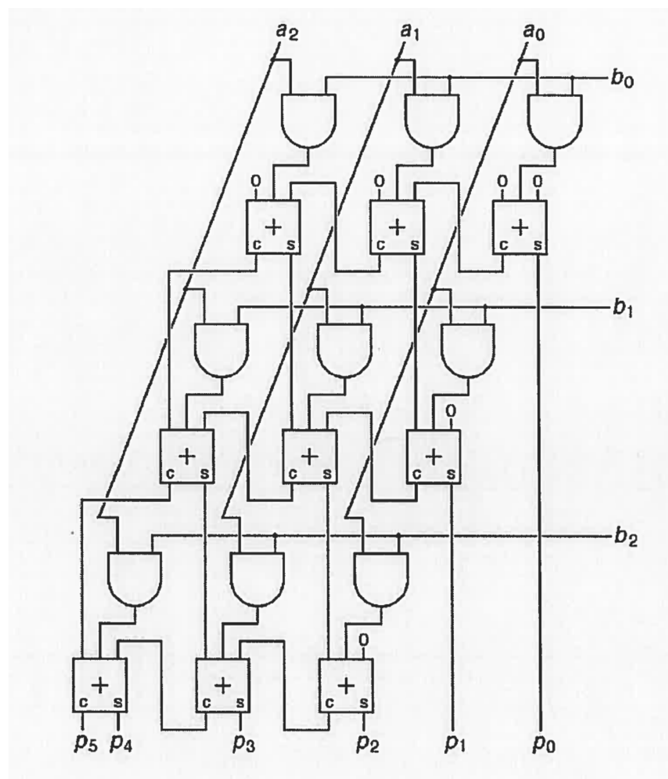
Läbiviidud katsed näitavad, et traditsioonilisel nn. „käskude eraldi testimise metoodikal" on oluliseks puuduseks see, et jäetakse arvesse võtmata rikked, mis on põhjustatud spetsiaalselt ALU juhtseadmes esinevate rikete poolt, mis võivad jääda maskeerituks, kui neid võimalikke rikkeid pole spetsiaalselt testimise objektidena arvesse võetud. Näiteks rike juhtseadmes võib tähendada seda, et mingi käsu aktiveerimine võib rikke tõttu aktiveerida lisaks ka veel teise käsu, aga võib jääda märkamatuks (maskeerituks). Juhtseadme testimine ei olnud aga antud töö eesmärgiks

5 Korrutaja testimine

Korrutaja on keeruline ja mahukas skeem, mille keerukus kasvab eksponentsiaalselt järkude arvu suurenemisega. Keerukuse kasvades aga kasvab ka eksponentsiaalselt testide genereerimiseks vajalik aeg. Siin on silmas peetud traditsioonilisi (deterministlikke) testide generaatoreid, mis ehitavad teste SAF tüüpi riketele.

Joonis 7. 4-bitiste arvude korrutamine

Sisuliselt korrutaja skeem koosneb mitmest lihtsummaatorite ridadest mis on omavahel ühe biti võrra nihkes. Siis on korrutatav vastavalt korrutaja bittidele summale õigete nihetega juurde liidetud. Tulemiks saab kaks korda pikem operand kui esialgsed sisendid.



Joonis 8. Lihtne korrutaja skeem

Pseudo-ammendavate testide genereerimise kohta korrutajate jaoks pole kirjanduses viiteid leida. Põhjus on ilmselt järgnevas. Kui summaatoreid realiseeritakse lineaarsete iteratiivsete 1-bitiste summaatorite abil, siis korrutajas on iteratiivsus maatrikstüüpi, kus 1-bitised lihtsummaatorid on üksteisega seotud väga keerulise struktuuriga. Igal lihtsummaatoril on üksteisega naabreid palju rohkem, kui traditsioonilises summaatori skeemis, mistõttu taolist pseudo-ammendava testi genereerimise lihtsalt tsüklist iteratiivset protsessi, nagu summaatori jaoks, pole korrutaja puhul võimalik rakendada.

5.1 Korrutamisseadme pseudo-ammendava testimise idee

Käesolevas töös ometigi rakendati selle ülesande lahendamiseks uut innovatiivset võtet – reasummaatorite järjestikkust „ekstraheerimist" korrutamisseadme maatriksist ja nende järjestikust tsüklilist testimist.

Selleks tuleb ühele operandile vastavatesse sisenditesse (A-sisend) saata mingi kindel kood (a_7, a_6, a_5, ... a_0), aga teisele operandile vastavasse sisendisse (B-sisend) nn. *„two-hot"* kood (analoogselt laialt kasutatavale *„one-hot"* koodile), kus kaks naaberbitti väärtusega 11 (kõigi ülejäänud bittide 0-väärtuste juures) läbivad tsükliliselt kogu koodi.

Need kaks bitti „selekteerivadki" korrutamisseadmes ühe konkreetse lihtsummaatorite rea, millele sisenditesse antakse korrutamisel seesama vektor (a_7, a_6, a_5, ... a_0), esimese liidetavana koos oma nihutatud kujuga – teise liidetavana. Niiviisi käivitataksegi selle korrutamisoperatsiooni tulemusena tegevusse see üksainus välja valitud 1-bitiste lihtsummaaatorite rida. Summaatorid kõigis eelnevates ridades töötavad selle heaks, et need kaks vajalikku liidetavat seadme primaarsisenditest aktiveeritud reasummaatorini jõuaksid. Sisuliselt on need summaatorid asetatud edastusrežiimi noorimate 0-bittide abil B-sisendis. Summaatorid kõigis järgnevates bittides töötavad aga selle nimel, et aktiveeritud summaatorite reaktsioonid testsignaalidele primaarväljunditeni jõuaksid. Need summaatorid on siis pandud signaalide levitamisrežiimi vanimate 0-bittide abil B-sisendis.

Iga rea summaatorite testimiseks on võimalik kasutada ühte ja sama ülal mainitud testvektorite komplekti, mis genereeritakse sarnaselt tavalise järjestikülekandega summaatori pseudo-ammendava testi genereerimise juhule.

Kahebitise 11-signaali tsüklilisel liikumisel läbi B-koodi teostatakse kaks operatsiooni: nihutatakse ühte ja sama testkoodi ühe reasummaatori sisenditest järgmise reasummatori sisenditesse ja asendatakse kõik ülejäänud summaatorite rea korrutaja seadmes signaalide läbimisrežiimi.

Testikoodi genereerimise ülesanne pseudo-ammendava testi genereerimiseks igale lihtsummaatorile korrutamisskeemis saab formuleerida järgmiselt: Leida n-bitise korrutamisseadme testimiseks, mille sisenditeks on (a_{n-1}, a_{n-2}, a_{n-1}, ... a_0) ja (b_{n-1}, b_{n-2}, b_{n-1}, ... b_0), testvektorite hulk T_j = {($a_{n-1,j}$, $a_{n-2,j}$, $a_{n-1,j}$, ... $a_{0,j}$), ($b_{n-1,j}$, $b_{n-2,j}$, $b_{n-1,j}$, ... $b_{0,j}$)} kus j => 8, nii et igale kombinatsioonile (C_{ij}, a_{ij}, b_{ij}) ∊ {000, 001, 010, 011, 100, 101, 110, 111}, kus ülekanded summaatorite vahel ühes ja samas reas arvutatakse valemitega

$c_i = c_{i-1} \oplus a_{i-1} \oplus b_{i-1}$, leiduks vähemalt üks testvektor T_j.

Kui sellised testvektorid õnnestub genereerida, et iga summaatori kõikide sisendite kombinatsioonid saaks läbi proovida, siis olekski korrutaja testimine sellega ära lahendatud. Kusjuures iga lihtsummaatori kõikide sisendite kombinatsioonide läbiproovimine tooks välja skeemi ükskõik mis vead, mitte ainult konstant tüüpi vead.

Ülesanne kujutab endast mahuka lineaarvõrrandite süsteemi lahendamist, mis aga oma regulaarsuse tõttu on isegi käsitsi kergesti lahendatav. Iga genereeritav testvektor tähendab iga 3-bitise lihtsummaatori sisendvektori puhul kahele bitile (ülekandele ja

eelmisele sisendbitile) kitsenduse kindlaks tegemist ja kolmandale „vabale” bitile väärtuse valimist nii, et oleks saavutatav soovitud konkreetne 3-bitine kood. Juhul, kui mingit konkreetset soovitud koodi pole võimalik kitsenduste tõttu genereerida, võib lahendamist alustada uue testvektori genereerimisega, kus antud summaatori sisenditele pole veel ühtegi kitsendust seatud.

Kogu ülesande lahendamine ehk siis vajalike testvektorite genereerimine toimub alates noorimatest järkudest suunaga vanemate järkude poole. Algul üritatakse iga summaatori puhul 8-vajaliku sisendkombinatsiooni saamiseks 8 vektori abil ülesanne lahendada, kui see ei õnnestu, lisatakse uus testvektor. Järgmiste järkude puhul saab „vaba” bitti kasutada vajalike koodide otsimisel ka juba seda uut testvektorit silmas pidades. Eesmärgiks on aga kasutada nii vähe kui võimalik tervikvektoreid, mis hõlmaksid endas kõik vajalikud 8 lokaalset 3-bitist vektorit iga lihtsummaatori sisendis.

Korrutamisseadme terviklik test sünteesitakse nüüd tsükliliselt, nii et iga A-sisendi jaoks sünteesitud vektor T_j oleks realiseeritud paaris iga B-sisendi jaoks sünteesitud testvektoriga (jooksev 11 läbi nullide, ehk two-hot koodide hulk).

Siit tuleneb ka lihtne pikkuse hinnang pseudoammendavale testile:

Testi pikkus = (n-1) * $|\{T_j\}|$,

kus $|\{T_j\}|$ märgib testvektorite hulka, mis sai genereeritud lihtsummaatorite lokaalsete testide genereerimise protsessis. Olgu märgitud, et kõikide osasummaatorite testimiseks kasutatakse ühte ja seda sama testvektorite hulka $|\{T_j\}|$.

Kokku võtlikult võiks nüüd öelda, et selline meetodi idee seisneb selles, et korrutaja operandis alati panna kaks kõrvuti olevat bitti 1-ks ja ülejäänud jätta 0-ks. Sellises olukorras peaks algoritmi järgi korrutatava väärtus nihkega ise endale juurde liituma, ehk aktiveerub üks sumaatorite rida korraga. Siis on sisuliselt tegemist summaatoriga, kus mõlemad operandid on tegelikult sama väärtusega, aga nihkes omavahel. Summaatoritele ammendavate testide koostamisel peab nüüd sellega arvestama. Tabelis (Tabel 3) on välja toodud üks selline varjant, kus igale summaatorile on koostatud ammendav test. Kollasega on märgitud vektorid mis moodustavad iga lihtsummaatori ammendava testi. Kuna operandid on omavahel seotud, siis ei õnnestu seekord piirduda vaid kaheksa vektori koostamisega, sest osadel summaatoritel tekivad korduvad testid. Selle probleemi saab lahendatud kui genereerida veel lisaks vektoreid, mis kataksid puudu olevad testid.

5.2 Eksperimendid MiniMIPS kombinatoorse korrutamisseadmega

Vaatluse all olev kombinatoorne korrutamisseade on võetud miniMIPS protsessori skeemist, millele on genereeritud pseudo-ammendavad testid eelpool kirjeldatud uue meetodi abil ja mille effektiivsust on hinnatud SAF tüüpi rikete simuleerimise ning katte arvutamise abil Turbo-Testeri rikete simulaatoriga.

Eksperimentide eesmärke oli neli: (1) kontrollida meetodi realiseeritavust, (2) genereerida vajalikud testvektorid, (3) hinnata nende kvaliteeti SAF rikete suhtes ja (4) püüda leida analoogilist korduvuse seaduspära, nagu see esineb traditsioonilise järjestikülekandega summaatori puhul, kus iga kahe järgu järel hakkavad testvektorid korduma.

Tabelis (Tabel 3) on välja toodud üks selline varjant, kus igale summaatorile on koostatud ammendav test. Kollasega on märgitud vektorid mis moodustavad iga lihtsummaatori ammendava testi. Kuna operandid on omavahel seotud, siis ei õnnestu seekord piirduda vaid kaheksa vektori koostamisega, nii nagu see õnnestus tavalise järjestikülekandega summaatori puhul. Põhjuseks on see, et iseseisva summaatori puhul moodustas kitsenduse lokaalse testi sünteesil ainult ülekande bitt, korrutaja puhul aga tekib igas järgus kaks kitsendust.

Tabel 3. Korrutaja pseudo-ammendava testi genereerimine

	C7	A7	A6	C6	A6	A5	C5	A5	A4	C4	A4	A3	C3	A3	A2	C2	A2	A1	A1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	1
3	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0
4	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1
5	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1
6	0	1	1	0	1	0	1	0	0	1	0	1	1	1	0	1	0	1	1
7	1	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1	1
8	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1
9	1	0	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	0	0
10	1	1	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1
																1	0	0	0
																1	1	0	0

Esimeses 0-järgus, kus ülekanne puudub, piisab lokaalsest 4-vektorilisest ammendavast testist 00, 01, 10 ja 11. Järgmise 1-järgu lihtsummaatorile ei ole võimalik kahte vektorit 100 ja 110 sünteesida vasturääkivate kitsenduste tõttu bitile A1. A1 väärtuseks nõutakse

0, aga A1=0 puhul ei teki ülekannet C2=1 järgmisse järku. 2. järgus leiame et 1. järgus loodud kitsendused ei võimalda 2. järgus enam konstrueerida kõiki vajalikke 8-t vektorit. Vektorite 011 ja 100 saamiseks tuleb genereerida lisaks 8-le veel kaks operandi. Neid 10-t operandi jooksvalt sünteesides igas alamtabelis (lihtsummaatorite lokaalseid vektoreid sünteesides) kahe kitsenduse juures õnnestus lõpuks saada 10 vektoriga kõigile lihtsummaatoritele (v.a. 1. järk) kõik vajalikud 8 lokaalset testi.

Tabeli 3 alusel saab nüüd kompileerida A-sisenditesse antavad 10 operandi (Tabel 4), milliseid kõiki tuleks kasutada ka kõigi B-sisenditesse antavate operandidega (Tabel 5).

Tabel 4. Testvektorid Korrutaja A-sisendile

Test-vektorid j	A-sisendid, bittide numbrid i							
	a7	a6	a5	a4	a3	a2	a1	a0
1	0	0	0	0	0	0	0	0
2	1	0	1	0	1	0	1	0
3	0	1	0	1	0	1	0	1
4	1	0	0	1	1	0	1	0
5	0	1	1	0	0	1	1	0
6	1	1	0	0	1	0	1	1
7	0	0	1	1	0	1	1	1
8	1	1	0	1	1	1	1	1
9	0	1	1	0	1	1	0	0
10	1	0	1	1	0	0	1	1

Tabel 5. Testvektorid korrutaja B-sisenditele

Test vektorid	B-sisendid, bittide numbrid i							
	b7	b6	b5	b4	b3	b2	b1	b0
1	0	0	0	0	0	0	1	1
2	0	0	0	0	0	1	1	0
3	0	0	0	0	1	1	0	0
4	0	0	0	1	1	0	0	0
5	0	0	1	1	0	0	0	0
6	0	1	1	0	0	0	0	0
7	1	1	0	0	0	0	0	0

Testvektorite koguarv korrutaja sisendisse, kasutades ülal toodud valemit on:

Testi pikkus = $(n-1) * |\{Tj\}| = 7 * 10 = 70$.

Vaadates tagasi summaatori testide genereerimise katsetele, siis SAF tüüpi rikete katte arvutamisel tekkis korrutaja testide puhul sarnane olukord. Käsitsi genereeritud pseudoammendavad testid ei anna täielikku rikete katte, nagu ka kiire ülekandega

summaatorite puhul. Summaatorite puhul oli põhjuseks summaatori regulaarsele osale lisatud kiire ülekande loogikaskeem, mille testimiseks pseudoammendavad testid ei ole mõeldud. Korrutaja puhul ei olnud töö tegemiseks antud aja jooksul pseudoammendava testi mitte 100%-lise SAF katte täpset põhjust võimalik välja selgitada. Küll aga tuleks loogiliselt järeldada, et korrutaja skeemis pidi esinema mingi mitteregulaarne osa. Olgu märgitud, et nimetatud skeem sai laboris sünteesitud SYNOPSYS projekteerimistarkvara abil.

Järgmiseks ülesandeks sai seetõttu analoogiliselt kiire summaatori katsetega proovida kokku panna genereeritud pseudo-ammendavad testid ja automaatselt genereeritud täiendavad testid, kasutades deterministlike testide generaatorit Turbo Tester.

Deterministlik testigeneraator üksinda töötades andis 100% rikete katte testiga, mis koosnes 40-st testvektorist. Käsitsi sai genereeritud 70-st testvektorist koosnev pseudoammendav test, mis katab ära 97% SAF tüüpi riketest. Pseudoammendava testide poolt mitteavastatud rikete avastamiseks genereeriti 12 täiendavat testvektorit, mis andsid kombineeritud testi pikkuseks 82 vektorit. Lõpuks sai läbi viidud veel üks katse, kus ühendati 40 deterministlikku testi ja 70 pseudoammendavat testi ning kasutati Turbo-Testri optimeerimisprogrammi, mis andis optimeeritud testi pikkuseks 28 testvektorit, mis katab kõik 100% korrutaja riketest

20. Kokkuvõttes tuleb aga öelda, et Turbo-Testri poolt genereeritud testid võimaldavad garanteerida üksnes SAF-tüüpi rikete katmist, samal ajal kui pseudoammendav test katab suvalisi rikkeid, kaasa arvatud ka suvalisi kordsete rikete konbinatsioone. Teiselt poolt, testide optimiseerimisprogrammi kautades saavutatakse küll testi pikkuse lühenemine, kuid samal ajal läheb ka kaduma pseudoammendava testi kvaliteet, mis tagab palju laiema rikete klassi katte.

Tabel 6. Eksperimendid 4 korrutajaga

Korrutaja bittide arv n	Rikete kate %			Testvektorite arv			Testide genereerimise ja simuleerimise aeg s		
	PET	DET	PET+D	PET	DET	PET+D	PET	DET	PET+D
4	95.87	100	100	30	27	34	0.001	0.001	0
8	97.97	100	100	70	40	82	0.002	0.012	0.009
16	97.0	99.97	99.97	150	73	185	0.07	0.48	0.39
32	96.5	99.8	99.98	310	134	381	2.73	6.25	2.41

21. Tabelis (Tabel 6) on toodud eksperimentide tulemused 4-, 8, 16- ja 32-bitiste korrutajate testimise kohta. Iga korrutaja kohta on esitatud SAF-tüüpi rikete katte protsent, testi pikkus (testvektorite arv) ja testide genereerimiseks ning simuleerimiseks vaja minev aeg (NB! Pseudoammendavad testid genereeriti käsitsi, konkreetsed testid kõigi korrutajate kohta on toodud Lisas).

6 Kokkuvõte

Käesolevas töös uuriti pseudoammendavate testide kasutamise efektiivsust mikroprotsessorite ALU struktuuride testimisel. Töö põhitulemuseks oli uudse innovatiivse meetodi välja töötamine korrutamisseadmele pseudoammendavate testide iteratiivseks genereerimiseks, mis teadaolevalt on esmakordselt väljapakutud idee korrutamisskeemide klassis.

Töö koosnes kolmest osast. Esimeses osas uuriti pseudoammendavate testide idee rakendamist ja saadavate testide kvaliteeti eri tüüpi summaatorite testimisel, katsetades nii järjestikülekandega kui ka kiire ülekandega summaatoreid. Teises osas laiendati funktsioonide ringi, mille puhul töötati välja pseudoammendavad testid, kasutades selleks vabavara mikroprotsessori MiniMIPS ALU-ga seotud käskude süsteemi. Kolmandas osas töötati esmakordselt välja pseudoammendavate testide genereerimise meetod korrutamisseadmele ja viidi läbi ulatuslik eksperimentide seeria 4-le eri suurusega korrutamisseadmele.

Pseudoammendava testi eeliseks on laiem rikete klassi katmine kui seda tagab klassikaline deterministlik testide genereerimine, kus eesmärgiks on traditsiooniliselt konstantrikete (SAF tüüpi rikete) katmine. Eksperimendid näitasid, et 100%-line SAF-tüüpi rikete kate on pseudoammendavate testidega tagatud üksnes täielikult regulaarse seadme (nt läbiva ülekandega summaatorite) puhul. Samal ajal aga täiendavate eksperimentidega sai näidatud, et igasugune anomaalia regulaarsusest (näiteks kiire ülekande loogikaskeemi lisamine summaatorisse) ei võimalda saada enam 100% SAF rikete katet. Küll aga on see saavutatav täiendavate vektorite lisamise teel, mida saaks genereerida deterministlike testide generaatoriga.

Kasutatud kirjandus

[1] Ubar, R., Digitaalsüsteemide diagnostika II Testide süntees ja analüüs http://www.pld.ttu.ee/~raiub/web_0103/diagnostika/loengukiled/ABIKS/%d5ppevahend_2.doc (22.05.2017)

[2] Ubar, R., Süsteemide diagnostika loengud http://www.pld.ttu.ee/~raiub/web_0103/diagnostika/loengukiled/ (22.05.17)

[3] Integrated Circuit Engineering Corporation, Yield and Yield Management, http://smithsonianchips.si.edu/ice/cd/CEICM/SECTION3.pdf (21.05.2017)

[4] Adder (electronics), (muudetud 02.05.2017) [WWW] https://en.wikipedia.org/wiki/Adder_(electronics) (21.05.2017)

[5] Carry-lokkahead adder,(muudetud 24.04.2017) [WWW] https://en.wikipedia.org/wiki/Carry-lookahead_adder (21.05.2017)

[6] Oyeniran, A. S., Ubar, R., Azad, S. P., Raik, J. High-Level Test Generation for Processing Elements in Many-Core Systems, Tallin University of Technology, Estonia

[7] Turbo-Tester, (muudetud 24.01.2008) [WWW] http://www.pld.ttu.ee/tt/ (16.05.2017)

[8] miniMIPS, [WWW] https://opencores.org/project,minimips (19.05.2017)

Lisa 1 – 4-bitiste summaatorite testide genereerimine

Tabel 7. 4-bitiste summaatorite testvektorid

4 bit adder								
a3	a2	a1	a0	cin	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0
0	1	0	1	0	0	1	0	1
1	0	1	0	1	1	0	1	0
0	0	0	0	1	1	1	1	1
1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1

	PE	D
total	114	114
covered	114	114
not covered	0	0
Coverage %	100	100
tests	8	12

4 bit fast adder 2 bit carry								
b3	b2	b1	b0	a3	a2	a1	a0	cin
0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0
0	0	0	0	1	1	1	1	0
0	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1
1	1	1	1	0	0	0	0	1
0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1

	PE	D	PE + D + optimize
total	130	130	130
covered	122	130	130
not covered	8	0	0
Coverage %	93.85	100	100
tests	8	15	10
optimized %			33.33

4 bit fast adder								
cin	b3	b2	b1	b0	a3	a2	a1	a0
0	0	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	0
0	0	0	0	0	1	1	1	1
0	0	1	0	1	0	1	0	1
1	1	0	1	0	1	0	1	0
1	1	1	1	1	0	0	0	0
1	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1

	PE	D	PE + D + optimize
total	164	164	164
covered	133	164	164
not covered	31	0	0
Coverage %	81.10	100	100
tests	8	34	16
optimized %			52.94

Lisa 2 – 8-bitiste summaatorite testide genereerimine

Tabel 8. 8-bitiste summaatorite testvektorid

8 bit adder																
a7	a6	a5	a4	a3	a2	a1	a0	cin	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D
total	226	226
covered	226	226
not covered	0	0
Coverage %	100	100
tests	8	15

8 bit fast adder 2 bit carry																
b7	b6	b5	b4	b3	b2	b1	b0	a7	a6	a5	a4	a3	a2	a1	a0	cin
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D	PE + D + optimize
total	258	258	258
covered	242	258	258
not covered	16	0	0
Coverage %	93.80	100	100
tests	8	19	12
optimized %			36.84

8 bit fast adder 4 bit carry																
cin	b7	b6	b5	b4	b3	b2	b1	b0	a7	a6	a5	a4	a3	a2	a1	a0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D	PE + D + optimize
total	326	326	326
covered	262	326	326
not covered	64	0	0
Coverage %	80.37	100	100
tests	8	39	26
optimized %			33.33

Lisa 3 – 16-bitiste summaatorite testide genereerimine

Tabel 9. 16-bitiste summaatorite testvektorid

16 bit ripple carry adder																																
a15	a14	a13	a12	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	cin	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D
total	450	450
covered	450	450
not covered	0	0
Coverage %	100	100
tests	8	17

16 bit fast adder 2 bit carry																																
b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	a15	a14	a13	a12	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0	cin
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D	PE + D
total	514	514	514
covered	482	514	514
not covered	32	0	0
Coverage %	93.77	100	100
tests	8	24	16
optimized %			33.33

16 bit fast adder 4 bit carry																																
cin	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0	a15	a14	a13	a12	a11	a10	a9	a8	a7	a6	a5	a4	a3	a2	a1	a0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

	PE	D	PE + D
total	650	650	650
covered	522	650	650
not covered	128	0	0
Coverage %	80.31	100	100
tests	8	49	32
optimized %			34.69

Lisa 4 – ALU testide genereerimine

Tabel 10. ALU funktsioonide testvektorid

ALU_new																	ATPG		total	
op1								op2								op		1.00		1264.00
a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	b0			156.00		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ADD				
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		covered	535.00	covered	535.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		not covered	729.00	not covered	729.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		Coverage %	42.33	Coverage %	42.33
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		tests	8.00	tests	8.00
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1					
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1					
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1					
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ADDU				
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		covered	519.00	covered	540.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		not covered	745.00	not covered	724.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1		Coverage %	41.06	Coverage %	42.72
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		tests	8.00	tests	16.00
0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1					
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	1					
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1					
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SUB				
0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	0		covered	447.00	covered	609.00
1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1		not covered	817.00	not covered	655.00
1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1		Coverage %	35.36	Coverage %	48.18
0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0		tests	8.00	tests	24.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0					
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0					
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0					
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SUBU				
0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	0		covered	441.00	covered	612.00
1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1		not covered	823.00	not covered	652.00
1	1	1	1	1	1	1	1	1	0	1	0	1	0	1	1		Coverage %	34.89	Coverage %	48.42
0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0		tests	8.00	tests	32.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0					
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0					
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0					

Lisa 5 – ALU testide genereerimine jätk

Tabel 11. ALU funktsioonide testvektorid jätk

ALU_new																				
																	ATPG	total		
op1								op2								op	1.00	1264.00		
a7	a6	a5	a4	a3	a2	a1	a0	b7	b6	b5	b4	b3	b2	b1	b0		156.00			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	AND	covered	228.00	covered	668.00
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		not covered	1036.00	not covered	596.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		Coverage %	18.04	Coverage %	52.85
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		tests	4.00	tests	36.00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	OR	covered	164.00	covered	711.00
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		not covered	1100.00	not covered	553.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		Coverage %	12.97	Coverage %	56.25
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		tests	4.00	tests	40.00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	XOR	covered	259.00	covered	759.00
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		not covered	1005.00	not covered	505.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		Coverage %	20.49	Coverage %	60.05
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		tests	4.00	tests	44.00
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	NOR	covered	244.00	covered	788.00
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1		not covered	1020.00	not covered	476.00
1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0		Coverage %	19.30	Coverage %	62.34
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		tests	4.00	tests	48.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	SLL	covered	187.00	covered	858.00
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		not covered	1077.00	not covered	406.00
																	Coverage %	14.79	Coverage %	67.88
																	tests	4.00	tests	50.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	SRL	covered	201.00	covered	910.00
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		not covered	1063.00	not covered	354.00
																	Coverage %	15.90	Coverage %	71.99
																	tests	4.00	tests	52.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	SRA	covered	194.00	covered	924.00
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		not covered	1070.00	not covered	340.00
																	Coverage %	15.35	Coverage %	73.10
																	tests	4.00	tests	54.00
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	LUI	covered	134.00	covered	942.00
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0		not covered	1130.00	not covered	322.00
																	Coverage %	10.60	Coverage %	74.53
																	tests	4.00	tests	56.00

Lisa 6 – Korrutaja testide genereerimine

Tabel 12. Korrutaja testide genereerimine

	C8	A8	A7	C7	A7	A6	C6	A6	A5	C5	A5	A4	C4	A4	A3	C3	A3	A2	C2	A2	A1	A1	A0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	1	0
3	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	1
4	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0
5	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	1	0
6	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1	0	1	0	1	1	1
7	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1
8	1	0	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1
9	1	1	0	1	0	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	0	0	0
10	1	1	1	1	1	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1
																			1	**0**	**0**	0	1
																			1	**1**	**0**	0	1

	C15	A15	A14	C14	A14	A13	C13	A13	A12	C12	A12	A11	C11	A11	A10	C10	A10	A9	C9	A9	A8
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0
3	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
4	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1
5	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0
6	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0
7	1	1	1	1	1	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1
8	1	0	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0
9	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1
10	0	0	0	1	0	0	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1

	C22	A22	A21	C21	A21	A20	C20	A20	A19	C19	A19	A18	C18	A18	A17	C17	A17	A16	C16	A16	A15
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1
3	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	0	1	0
4	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1
5	1	0	1	1	1	1	1	1	0	1	0	1	0	1	1	0	1	0	1	0	0
6	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0	1	0	1
7	1	1	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1
8	1	1	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	0
9	0	1	1	0	1	0	1	0	0	1	0	1	1	1	1	1	1	0	1	0	1
10	1	0	1	1	1	0	1	0	1	1	1	0	1	0	1	0	1	1	0	1	0

	C29	A29	A28	C28	A28	A27	C27	A27	A26	C26	A26	A25	C25	A25	A24	C24	A24	A23	C23	A23	A22
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0
3	0	0	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1
4	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1	0	1	0
5	0	1	1	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	1	0	0
6	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1	0	1	1
7	0	1	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	0	1	0	1
8	1	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	1	1	1	1	1
9	1	0	1	1	1	0	1	0	1	0	1	1	0	1	0	1	0	0	1	0	1
10	1	1		1		1	1	1	0	1	0	1	1	1	0	1	0	1	1	1	0

	C31	A31	A30	C30	A30	A29
1	0	0	0	0	0	0
2	0	1	0	0	0	1
3	0	1	1	0	1	0
4	0		0	1	0	0
5	1	0	0	1	0	1
6	0	0	1	0	1	0
7	1	0	1	0	1	1
8	1	1	1	1	1	1
9	1		1	1	1	0
10	1	1	0	1	0	1

Lisa 7 – Korrutaja A-operandi vektorid

Tabel 13. Korrutaja A-operandi vektorid

	32-bit mult																16-bit mult								8-bit mult				4-bit mult			
A	3	3	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
3	1	1	0	1	0	0	1	1	0	1	0	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
4	**0**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	0
5	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
6	0	1	0	0	1	1	0	0	1	1	0	1	0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	1	1
7	0	1	1	0	0	1	1	1	0	1	1	1	0	0	1	1	1	1	1	1	0	0	1	1	0	0	1	1	0	1	1	1
8	1	1	1	1	1	1	0	1	1	1	0	1	1	1	0	1	0	1	0	1	1	1	1	0	1	1	0	1	1	1	1	1
9	**0**	1	0	1	0	1	1	0	0	1	1	0	0	1	1	0	1	0	1	1	1	0	1	1	0	1	1	0	1	1	0	0
10	1	0	1	**0**	1	0	1	0	1	0	1	0	1	0	1	1	0	0	0	1	0	1	0	1	1	0	1	1	0	0	1	1

Lisa 8 – Korrutaja B-operandi vektorid

Tabel 14. Korrutaja B-operandi vektorid

	32-bit mult																16-bit mult								8-bit mult				4-bit mult			
B	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0																																
21	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
24	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0