Juhani Laasanen

VHDL-ohjelman testaaminen geneettisellä algoritmilla

Tekniikan ja innovaatiojohtamisen yksikkö
Diplomityö
Automaatiotekniikka

Vaasa 2023

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Juhani Laasanen
Tutkielman nimi: VHDL-ohjelman testaaminen geneettisellä algoritmilla
Tutkinto: Diplomi-insinööri
Oppiaine: Automaatiotekniikka
Työn valvoja: Yliopistonlehtori Timo Mantere
Työn ohjaaja: Apulaisprofessori Petri Välisuo
Valmistumisvuosi: 2023 **Sivumäärä:** **91**

TIIVISTELMÄ:
Tämä diplomityö käsittelee geneettisten algoritmien soveltamista ohjelmistotestauksessa. Työn tavoitteena oli tuottaa VHDL-ohjelman testaukseen testitapauksia ja tutkia geneettisen algoritmin soveltuvuutta testitapauksien tuottamisessa Intelin SoC FPGA -ympäristössä.

Geneettisen algoritmin soveltuvuus ohjelmistotestaukseen riippuu pitkälti siitä, mitä halutaan testata ja millä laajuudella. Työssä pohdittiin eri testausvaihtoehtoja ja niiden käyttämistä SoC FPGA:ssa. Tutkimukseen valittiin haarakattavuusmenetelmä, sillä se on riittävän kattava luotettavan testaustuloksen saamiseksi. Tässä työssä geneettisen algoritmin tehtävänä oli etsiä minimimäärä testitapauksia, joilla ohjelman kaikki haarat saatiin käytyä läpi. Lisäksi haarakattavuusmenetelmässä testitapauksen kulkua ohjelmassa on yksinkertainen seurata ja siitä saatiin helposti palaute geneettisen algoritmin hyvyysfunktioon.

Työssä ohjelmoitiin C-kielellä geneettinen algoritmi, jolla voidaan testata erinäisiä VHDL-prosesseja alustan logiikkapuolella. Työssä testattiin viittä eri ohjelmaa sekä geneettisellä algoritmilla että satunnaisgeneraattorilla. Molempien tulosten tutkimisen jälkeen havaittiin, että useimmissa testattavissa ohjelmissa geneettinen algoritmi löysi pienellä vaivalla minimimäärän testitapauksia, joilla kaikki haarat saatiin käytyä läpi. Esimerkiksi viidennessä testattavassa ohjelmassa satunnaisgeneraattorin tulokset vaihtelivat 27:n ja 93:n testitapauksen välillä. Samassa tapauksessa geneettinen algoritmi löysi aina minimimäärän eli viisi testitapausta. Poikkeuksena geneettisen algoritmin tehokkuudelle oli yksinkertaisin testattava ohjelma, jossa haaroja oli melko pieni määrä ja kromosomin pituus pieni, jolloin algoritmi joskus juuttui lokaaliin maksimiin. Satunnaisgeneraattorin tulokset vaihtelivat kolmen ja kuuden välillä, kun geneettinen algoritmi sai tuloksiksi vaihtelevasti kaksi ja kolme testitapausta.

AVAINSANAT: geneettinen algoritmi, SoC FPGA, ohjelmistotestaus, lasilaatikkotestaus, haarakattavuus

UNIVERSITY OF VAASA
School of Technology and Innovations
Author: Juhani Laasanen
Topic of the Thesis: Testing a VHDL Program with a Genetic Algorithm
Degree: Master of Science in Technology
Major of Subject: Automation Technology
Supervisor: University Lecturer Timo Mantere
Instructor: Associate Professor Petri Välisuo
Year of Completing The Thesis: 2023 **Pages:** **91**

TIIVISTELMÄ:
This thesis deals with the application of genetic algorithms in software testing. The purpose of the study was to generate test cases for testing a VHDL program and to investigate the suitability of a genetic algorithms for generating test cases in Intel SoC FPGA environment.

The suitability of a genetic algorithm for software testing depends largely on what is to be tested and to what extent. There were many options for testing considered, especially from the point of view of how they work with the SoC FPGA, but in the end the branch coverage method was selected, since its coverage is large enough for getting a reliable testing result. In the study the purpose of the genetic algorithm was to find the minimum number of test cases required to cover all possible branches of the program. Additionally, the branch coverage method makes it easy to track the test case progress in the program and provides feedback to the genetic algorithm's fitness function.

A genetic algorithm was programmed in C language to test various VHDL processes on the platform's logic side. In the study, five different programs were tested using both the genetic algorithm and a random generator. After analyzing the results of both methods, it was found that the genetic algorithm easily found the minimum number of test cases required to cover all the branches in most of the programs tested. For example, in the fifth program, the results from the random generator varied between 27 and 93 test cases, while the genetic algorithm always found the minimum number, which was five test cases. The only exception to the effectiveness of the genetic algorithm was the simplest program tested, which had a small number of branches and a short chromosome length, causing the algorithm to sometimes get stuck in a local maximum. In this case, the results from the random generator varied between three and six, while the genetic algorithm produced results that varied between two and three test cases.

KEYWORDS: genetic algorithm, SoC FPGA, software testing, white box testing, branch coverage

Sisällys

Lyhenteet

ARM	Advanced RISC Machine (katso RISC)
aa	ehdotettu kromosomi
b	geneettisen algoritmin ehdottaman polun haarojen vektori
b1	käytyjen haarojen vektori
bb	uusien haarojen lukumäärä
cc	läpikäytyjen haarojen lukumäärä
CFG	Control Flow Graphic
CPLD	Complex Programmable Logic Device
f	hyvyysfunktio
FPGA	Field Programmable Gate Array
GA	Geneettinen algoritmi
HDL	Hardware Description Language
HPS	Hard Processor System
i	binääriluvun indeksi
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
j	haaran indeksi
LXDE	Lightweight X11 Desktop Environment
m	haarojen lukumäärä
MMC	MultiMediaCard
N	desimaaliluku
n	binääriluvun bittien lukumäärä
P	populaatio
PLD	Programmable Logic Device
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTL	Register Transfer Level
SD	Secure Digital

SDRAM	Synchronous Dynamic Random Access Memory
SOC	System On Chip
SOPC	System On Programmable Chip
SPI	Serial Peripheral Interface
VHDL	Very High Speed Integrated Circuit HDL (katso HDL)
VLSI	Very Large Scale Integration
x	numero indeksin kohdalla

ALKULAUSE

Tämä diplomi-insinöörin tutkintoa varten tehty työ on toteutettu Vaasan yliopiston tekniikan ja innovaatiojohtamisen yksikössä.

Haluan kiittää työn alkuperäistä ohjaajaani professori Jarmo Alanderia diplomityön mielenkiintoisesta aiheesta sekä neuvoista. Lisäksi haluan kiittää TkT Petri Välisuota avusta ja vinkeistä FPGA-laitteen käytössä sekä lopullista työn tarkastajaa yliopistonlehtori Timo Manteretta.

1 Johdanto

Ohjelmistotestaus on ohjelmistotuotannon vaihe, jossa varmistetaan, että ohjelma toimii, kuten sen on tarkoitus. Testauksen tavoitteena on siis löytää mahdolliset virheet ja varmistettava ohjelman toimivuus. Yleensä testaukseen käytetään valmiita testausohjelmia, jotka ajavat ohjelmistoa annettujen kriteereiden mukaisesti. (Hetzel 1988: 3–5.)

Tämän diplomityön tavoitteena oli luoda SoC FPGA:lle (System on Chip Field Programmable Gate Array) C-kielinen geneettisellä algoritmilla toimiva testausohjelma, joka minimoi testitapauksien määrän testattavaan VHDL (Very High Speed Integrated Circuit Hardware Description Language) -ohjelmaan. Lisäksi tarkoituksena oli tutkia geneettisen algoritmin soveltuvuutta ohjelmistotestaukseen Intelin SoC FPGA -ympäristössä. Geneettisen algoritmin kommunikointiin FPGA-järjestelmän kanssa käytettiin valmiita kommunikointirakenteita, jotka muokattiin tämän diplomityön tarpeisiin sopiviksi.

1.1 Geneettinen algoritmi ohjelmistotestauksessa

Geneettistä algoritmia ohjelmistotestauksessa on tutkittu runsaasti. Testauksen apuna käytetyistä geneettisistä algoritmeista löytyy noin 400 tieteellistä julkaisua, joista noin 100 käsittelee ohjelmistotestausta (Alander 2014). Geneettisiä algoritmeja voidaan käyttää lukuisilla eri tavoilla ohjelmistotestauksen alalla. Niillä voidaan esimerkiksi optimoida eri testaustapoja paremmiksi tai tuottamaan testitapauksia testattavalle ohjelmistolle.

Esimerkiksi Glamorganin yliopistossa on tehty väitöskirja, jossa tutkitaan geneettisellä algoritmilla luotuja testitapauksia ohjelmistotestaukseen (Sthamer 1996). Tutkimuksessaan Sthamer vertaili satunnaisgeneraattorin ja geneettisen algoritmin luomia testitapauksia. Hän totesi, että geneettinen algoritmi löytää testitapaukset, joilla saavutetaan riittävä testikattavuus paljon satunnaisgeneraattoria tehokkaammin. Ainoana heikkoutena geneettiselle algoritmille Sthamer mainitsee, että se vaatii tietokoneelta runsaasti laskentatehoa. Tähän voidaan todeta, että tietokoneiden laskentateho kasvaa jatkuvasti, mutta samalla myös ohjelmien monimutkaisuus.

Toinen esimerkki testitapausten tuottamisesta on Vaasan yliopistossa tehty tutkimus, jossa Timo Mantere pyrki etsimään testitapauksia, joilla testattavasta ohjelmasta löydettäisiin virhetilanteita (Mantere 2003). Tutkimuksessaan Mantere loi testitapauksia erilaisille ohjelmille, kuten kuvankäsittelyfiltterille sekä mittausohjelmalle. Testitapaukset eivät siis hae maksimaalista testikattavuutta, vaan ohjelmaa kuormitetaan musta laatikko-testaustavalla. Saatujen tulosten perusteella Mantere toteaa, että geneettinen algoritmi on satunnaisgeneraattoriin verrattuna tehokkaampi löytämään testitapauksia, jotka voivat aiheuttaa ohjelmassa virhetilanteen.

Testauksen optimoinnista on olemassa esimerkiksi Praveen Ranjan Srivastavan tutkimus, jossa pyrittiin optimoimaan testauksen tehokkuutta. Tutkimuksessaan Srivastava käytti geneettisiä algoritmeja etsiäkseen ohjelmasta kriittisiä osia, jotka tulisi testata ensimmäisenä. Näin vältyttäisiin kattavasta testaukselta, mikäli jo kriittisissä osissa esiintyisi virheitä. (Srivastava 2009.)

Geneettisiä algoritmeja käytetään paljon myös VLSI (Very Large Scale Integration) -piirien testauksessa, kun yritetään löytää tehokkaampia tapoja testata piirejä. Geneettinen algoritmi voi auttaa löytämään optimaalisia testausparametreja, kuten testitapauksia tai testitapausten minimimääriä. Näillä testitapauksilla pyritään usein havaitsemaan piirien toimintahäiriöitä. (Mazumer & Rudnick 1998.)

Monessa testattavassa ohjelmassa, missä haetaan suurta testikattavuutta, ei kuitenkaan ole mahdollista saavuttaa haluttua testikattavuutta vain yhdellä luodulla testitapauksella. Tämän vuoksi onkin tärkeää, että testattavan ohjelmiston jo läpikäytyä osaa voidaan seurata, jotta uudet testitapaukset saadaan kohdistettua ohjelmiston osaan, jota ei ole vielä käyty läpi. (Roper, Maclean, Brooks, Miller & Wood 1995.)

1.2 Diplomityön rakenne

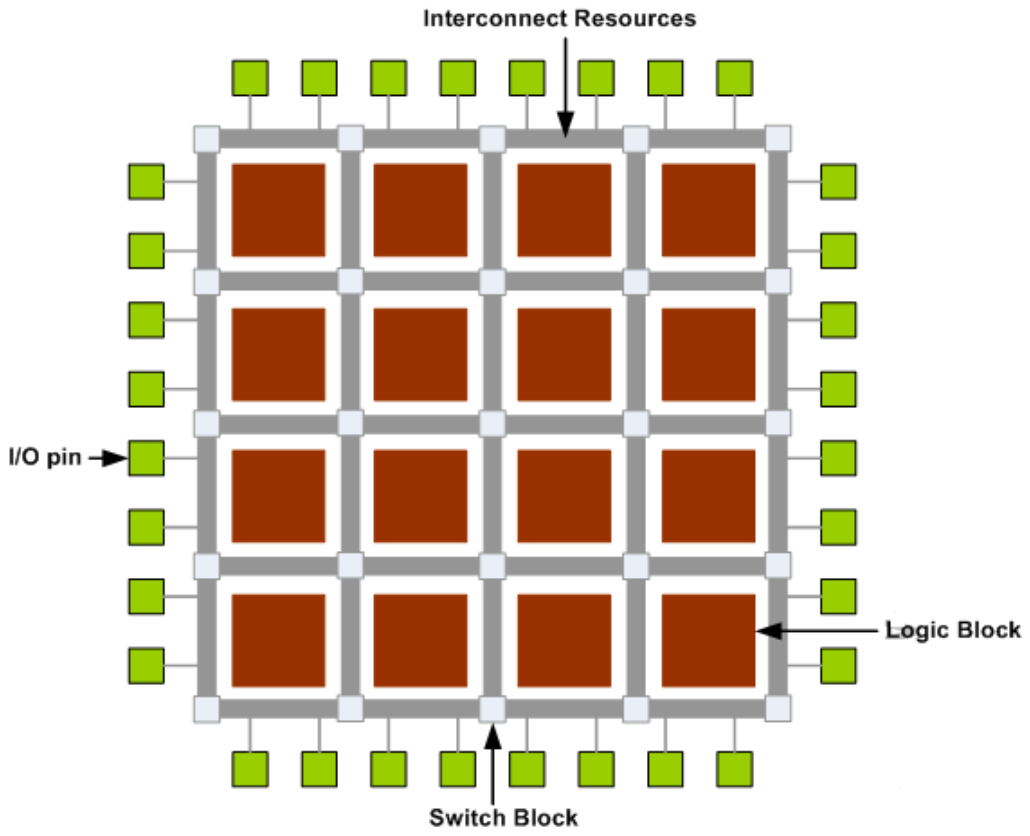
Työn toisessa luvussa käydään läpi FPGA:n toimintaperiaatteita, kehityksen historiaa sekä sen etuja ja eroja muihin tekniikoihin verrattuna. Lisäksi siinä tarkastellaan SoC

FPGA:n tekniikkaa ja sen eroa tavalliseen FPGA:han verrattuna. Luvussa esitellään myös alustan toimintaa siten, että toiminnasta tarkastellaan esimerkiksi alustan eri järjestelmien kommunikointia, joka on sen toiminnan ja ohjelmoinnin kannalta yksi tärkeimpiä asioita.

Kolmannessa luvussa käydään läpi lyhyesti ohjelmistotestausta, tarkastellaan muutamia eri testaustapoja ja esitellään eri kattavuuskriteereitä. Neljännessä luvussa käydään läpi diplomityön testausjärjestelmä. Geneettinen algoritmi on ohjelmoitu C-kielellä alustan prosessoripuolelle. Testattava ohjelma sijaitsee alustan FPGA-puolella. Viidennessä luvussa esitellään testattavat ohjelmat ja tarkastellaan testauksen soveltamista niihin. Lisäksi luvussa käydään läpi tulokset jokaisen testattavan ohjelman kohdalla, jolloin geneettisellä algoritmilla saatuja tuloksia verrataan satunnaisgeneraattorin antamiin tuloksiin. Kuudennessa luvussa pohditaan testauksen onnistumista, hyödyllisyyttä sekä mahdollista jatkokehitystä.

2 Field Programmable Gate Array

PLD eli Programmable Logic Device on ohjelmoitava logiikkapiiri. Ohjelmoitavia logiikkapiirejä on useita erilaisia, joista yksinkertaisin on SPLD (Simple Programmable Logic Device) ja monimutkaisempi on FPGA (Field Programmable Gate Array). Kun yksinkertaisempi logiikkapiiri voi sisältää esimerkiksi sisääntulona vain AND-portteja, eroaa FPGA siitä rakenteellisesti siten, että se sisältää suuren määrän geneerisiä logiikkaelementtejä. Niiden väliset kytkennät voidaan uudelleenohjelmoida. Tätä logiikkaelementtipiiriä ympäröi kehä I/O-elementtejä, jotka voidaan myös uudelleenohjelmoida (Kuva 1). Tämä on suuri etu, sillä vaikka rakenne on yksinkertainen ja edullinen, se on silti yksinkertaista logiikkapiiriä paljon monipuolisempi. (Chu 2008: 11–12; Wilson 2007: 5–7.)



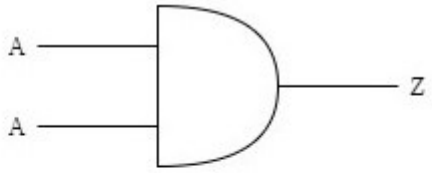
Kuva 1. Geneerinen FPGA-arkkitehtuuri, jossa logiikkaelementtipiiriä ympäröi kehä I/O-elementtejä (Kulov 2014.)

2.1 Hardware Description Language

FPGA:ta ohjelmoidaan käyttämällä laitteistonkuvauskieltä eli Hardware Description Languagea (HDL), joka käyttää tekstiä kuvaamaan digitaalista logiikkapiiriä tai järjestelmää. HDL perustuu kolmeen eri tasoon. Korkeimmalla tasolla määritetään järjestelmän arkkitehtuuri eli algoritmin funktiot ja kuinka ne ovat yhteydessä toisiinsa. Arkkitehtuurin alapuolella oleva taso on Register Transfer Level (RTL), jossa määritellään tiedon liikkuminen, muisti sekä tietoon kohdistuvat loogiset operaatiot. Viimeisenä tasona on loogisten porttien toteutus transistoreiksi. Nykyään teollisuudessa ja akateemisessa maailmassa yleisesti käytössä olevat kielet ovat Verilog sekä VHDL, jotka molemmat ovat IEEE-standardeja. (Grout 2008: 193.)

2.2 Very High Speed Integrated Circuit HDL

Very High Speed Integrated Circuit HDL (VHDL) on Ada-ohjelmointikieleen perustuva laitteistonkuvauskieli, joka sai alkunsa vuonna 1980 Yhdysvaltojen puolustusministeriön toimesta. Kielessä on kolme kuvaustapaa: rakenne (structural), tietovuo (dataflow) sekä käyttäytyminen (behaviour). Ohjelmoinnissa voidaan käyttää yhtä näistä kuvaustavoista tai niiden yhdistelmää. Rakennemallissa kuvataan järjestelmän rakennetta porttien sekä niiden kytkentöjen avulla. Tietovuomallissa kuvataan tiedon liikkumista sisääntulosta ulostuloon sekä signaalien välillä. Käyttäytymismallissa kuvataan järjestelmän käyttäytymistä algoritmeilla. (Zwolinski 2004: 68–69, 117.) Seuraavassa esimerkissä on yksinkertainen AND-veräjä (Kuva 2) sekä sitä kuvaava VHDL-kielinen ohjelma:




```
library IEEE;
use IEEE.std_logic_1164.all;

-- entity-osassa määritetään sisäänmenot ja ulostulot
entity And_Gate is
port ( A : in std_logic;
               B : in std_logic;
               Z : out std_logic);
end And_Gate;

--architecture-osassa määritellään toteutus
architecture Dataflow of And_Gate is
begin
     Z <= A and B;
end Dataflow;
```

Kuva 2. Yksinkertainen AND-veräjä, jossa kaksi sisääntuloa ja yksi ulostulo sekä sen VHDL-kielinen koodi.

2.3 Verilog

Toinen nykyään laajasti käytetty ohjelmointikieli on Verilog, jonka Gateway Design System Corporation julkaisi vuonna 1983. Erona VHDL-kieleen on, että Verilogin muistuttaessa enemmän C-kieltä, on VHDL Ada-pohjainen. Lisäksi Verilog on laitteistoläheisempi, jolloin sillä on VHDL:ää helpompi mallintaa logiikkapiirejä transistoritasolla. Toisaalta VHDL:llä on helpompi kuvata järjestelmää käyttäytymismallilla. Jotkut simulointiohjelmat hyväksyvät molemmat kielet, jolloin on mahdollista käyttää molempia kieliä esimerkiksi siten, että käytetään VHDL:ää korkean tason mallinnukseen ja Verilogia matalan tason mallinnukseen. (Wilson 2007: 11–12; Zwolinski 2004: 327.) Verilogin pohjalta on kehitetty myös ohjelmointikieli SystemVerilog, joka on erityisesti suunniteltu testaukseen. Sen avulla voidaan myös suunnitella Verilogiin nähden entistä monimutkaisempia järjestelmiä (Ashenden 2008: 22.) Seuraavassa esitellään AND-veräjän Verilog-kielinen ohjelma:

```
module AND(output Y, input A, B);
    and(Y, A, B);
endmodule
```

2.4 Quartus Prime

Intel Quartus Prime on alun perin Alteran kehittämä logiikkapiirien suunnitteluohjelma, jolla voidaan ohjelmoida VHDL- ja Verilog-kielillä. Lisäksi siinä on visuaalinen suunnittelu, piirien simulointi sekä muita ominaisuuksia riippuen käytettävästä versiosta. Peruskäyttäjille on olemassa ilmainen kevyempi versio (Lite), joka sisältää rajoitetun määrän tuettuja laitteita sekä ohjelman lisäominaisuuksia. Ohjelman maksulliset versiot on tarkoitettu lähinnä oppilaitoksille sekä yrityksille, ja niiden hinnat ovatkin useita tuhansia dollareita. (Intel 2018e).

2.5 SoC FPGA

System on a Chip (SoC) tai System on Programmable Chip (SoPC) on mikropiiri, joka sisältää suorittimen, muistia sekä muita mahdollisia ominaisuuksia yhdessä piirissä. Mikropiiriä, joka sisältää suorittimen lisäksi FPGA-logiikkapiirin kutsutaan SoC FPGA:ksi (Chu 2011: 4–5.)

Altera, joka on yksi suurimmista FPGA-piirien valmistajista, aloitti SoC FPGA -piirien tuotannon vuonna 2012. Sen ensimmäinen myyty piiri sisälsi ARM Cortex-A9-suorittimen sekä Cyclone V -FPGA-piirin (Maxfield 2012). Vuonna 2015 Altera myytiin Intelille, josta tuli kaupan myötä maailman toiseksi suurin uudelleenohjelmoitavien mikropiirien valmistaja (Clark 2015).

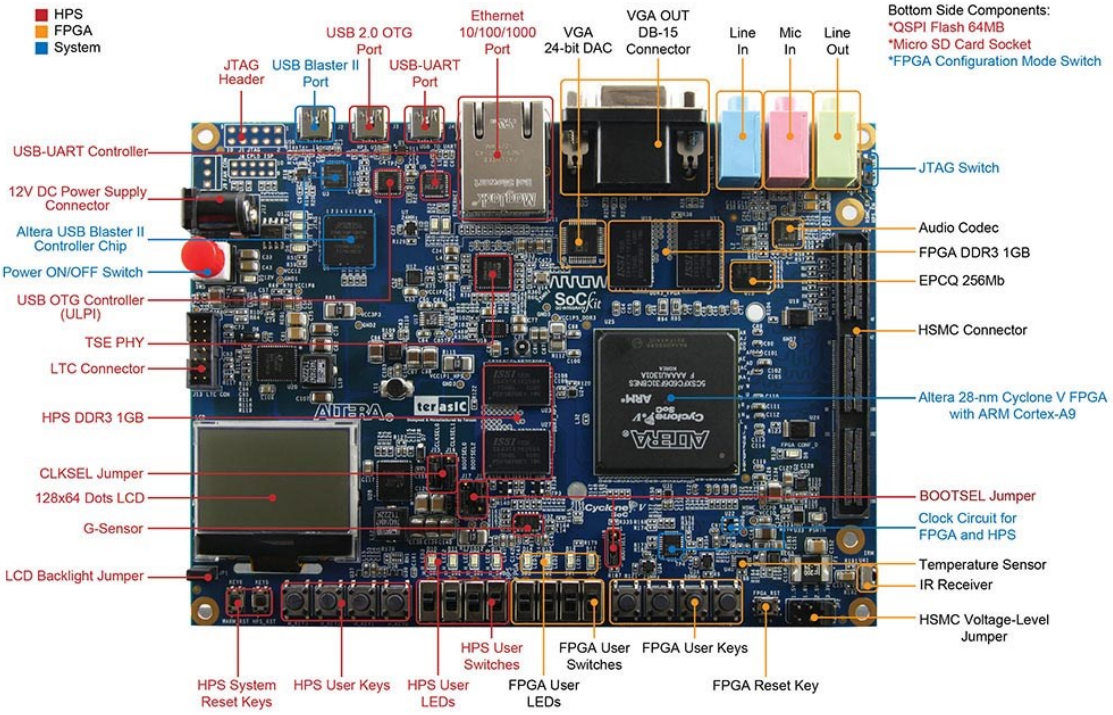
2.5.1 Rakenne

Vaikka prosessori ja logiikkapiiri ovat samassa alustassa, ovat ne silti kaksi erillistä järjestelmää. Prosessori sijaitsee HPS-järjestelmässä (Hard Processor System) ja logiikkapiiri omassa FPGA-fabric-järjestelmässään. Järjestelmät ovat keskenään riippuvaisia toisistaan, mutta niitä voidaan käyttää myös osittain erikseen. Esimerkiksi HPS-puoli voidaan käynnistää ja sitä voidaan käyttää riippumatta siitä, onko FPGA käynnissä. Toisaalta FPGA:ta ei voi käynnistää ilman HPS:n käynnistystä. Kumpaankin järjestelmään on lisäksi alustassa sijoitettu erillisiä vipuja ja painikkeita. Tämän lisäksi molemmilla järjestelmillä on erilliset I/O-liitännät. (Intel 2018a.)

2.5.2 Altera Cyclone V SoC FPGA

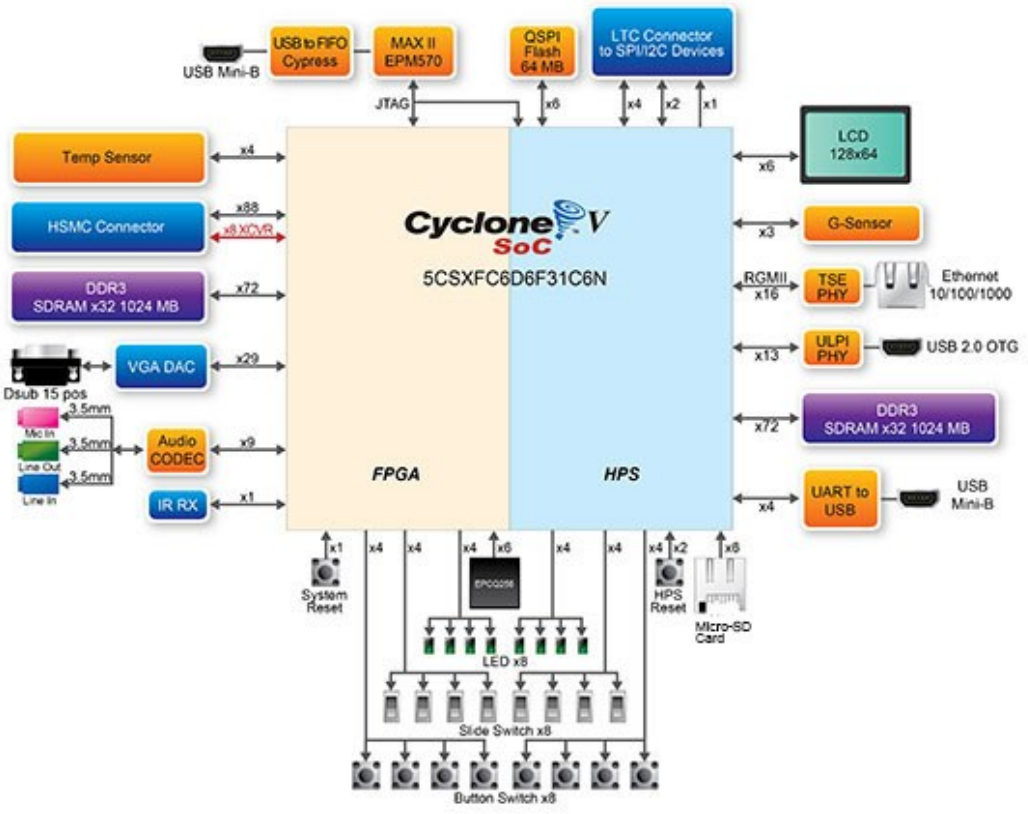
Cyclone V on Alteran Cyclone FPGA-tuoteperheen viidennen sukupolven alusta. Alustasta on useita eri versioita, joista osa toimii itsenäisesti ja osa on integroitu Arm-prosessorin kanssa (SoC FPGA). (Intel 2018b.)

Diplomityössä käytetty alusta on Terasicin SoCkit, jossa on yhdistetty Cyclone V SX SoC FPGA-piiri sekä Dual-Core Arm Cortex-A9-prosessori. Siinä on 110 000 ohjelmoitavaa logiikkaelementtiä, sekä 5 140 kilobittiä sulautettua muistia. Lisäksi siinä on 2 gigabittiä DDR3-muistia, josta toinen puoli on HPS:n ja toinen FPGA:n käytössä, sekä mahdollisuus USB-, Ethernet-, VGA- ja audioliitännöille sekä paikka Micro SD-kortille. (Terasic 2018.) Kuva 3 esittää käytetyn alustan, johon on merkitty komponentit ja liitännät.



Kuva 3. Altera Cyclone V SoC FPGA, johon on merkitty sen komponentit ja liitännät. Punaisella on merkitty HPS-puolen, keltaisella FPGA-puolen ja sinisellä järjestelmän omat komponentit ja liitännät (Terasic 2018).

Kuva 4 esittää käytetyn alustan lohkokaavion. Kaaviossa näkyy laitteen rakenne yleisellä tasolla ilman kuvia komponenteista. Kaaviosta näkyy eri osien liitännät eri järjestelmiin. Esimerkiksi kuvan alaosasta nähdään, että HPS- ja FPGA-puolella on kummallakin neljä LED-valoa sekä neljä kytkintä.



Kuva 4. Lohkokaavio alustasta. Sinisellä on merkitty liitännät, violetilla muistit sekä keltaisella sensorit, lähetin-vastaanotin-piirit, flash-muistit ja CPLD-piiri. Kaaviossa näkyy laitteen rakenne yleisellä tasolla, sekä eri komponenttien kytkeytymiset eri järjestelmiin (Terasic 2018).

2.5.3 Hard Processor System

HPS eli Hard Processor System on SoC FPGA:n osajärjestelmä, joka sisältää kaiken itsenäiseen toimintaan tarvittavan laitteiston. Tähän laitteistoon kuuluu esimerkiksi RAM-, sekä ROM-muistit, SD- (Secure Digital) ja MMC-korttien (MultimediaCard) kontrollerit, USB-liittimet (Universal Serial Bus) sekä SPI-kontrollerit (Serial Peripheral Interface). (Intel 2019).

Näiden lisäksi HPS sisältää ARM-mikroprosessorin. ARM on Arm Holdings-yhtiön mikroprosessoriarkkitehtuuri, joka on erittäin suosittu esimerkiksi matkapuhelimissa sekä sulautetuissa järjestelmissä. Esimerkiksi vuonna 2007 matkapuhelimista jopa 90% käytti ARM-prosessoria. (Arm 2007). ARM Cortex A-sarja on suunniteltu monimutkaisiin

tehtäviin, kuten tukemaan monipuolisesti eri sovelluksia ja käyttöjärjestelmiä. A-sarjasta Cortex-A9 on yksi laajimmin käytetyistä, sillä se sopii hyvin vähän virtaa kuluttaviin ja kohtuullisen hintaisiin laitteisiin, kuten SoC FPGA:han. (Arm 2018.)

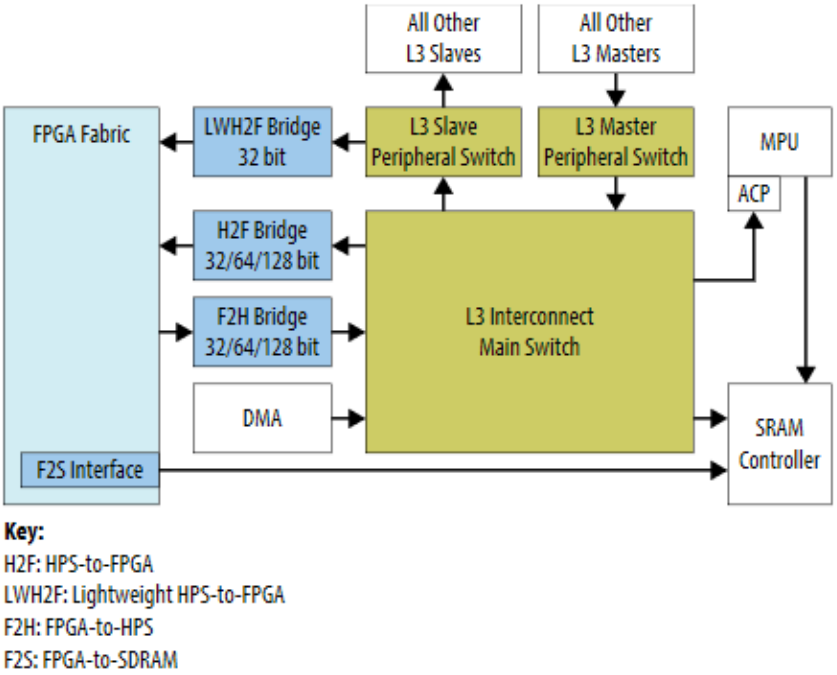
Järjestelmässä on myös oma graafinen käyttöliittymä eli työpöytäympäristö, joka tässä laitteessa on LXDE (Lightweight X11 Desktop Environment). LXDE on C-kielellä ohjelmoitu graafinen, Linux-pohjainen kevyt järjestelmä, joka käyttää vain vähän muistia sekä prosessoritehoa. Tämän vuoksi se sopii hyvin pieniin laitteisiin. Työpöytäympäristön kautta voidaan ohjelmoida sekä ajaa esimerkiksi C-pohjaisia ohjelmia. Lisäksi, vaikka FPGA:ta ohjelmoidaan esimerkiksi Quartus Prime-ohjelmalla, voidaan työympäristön kautta siirtää ja ajaa FPGA:lle tehdyt ohjelmat. (LXDE 2018.)

2.6 Järjestelmien välinen kommunikointi

Järjestelmien väliseen kommunikointiin on tarjolla erilaisia rajapintoja. Yksi näistä rajapinnoista on Intelin tarjoama Avalon Memory Mapped Slave (AMMS), jonka avulla järjestelmien välillä voidaan kommunikoida muistiosoitteiden kautta. AMMS muodostaa järjestelmien välille väylän, joka *näyttäytyy* FPGA:lle muistina. Tällöin HPS:n puolelta voidaan kommunikoida suoraan muistiosoitteisiin ja FPGA:n puolelta näennäisesti. Todellisuudessa FPGA:n puolelta ollaan yhteydessä itse väylään. (Intel 2018c.) Tässä diplomityössä käytetään kyseistä rajapintaa.

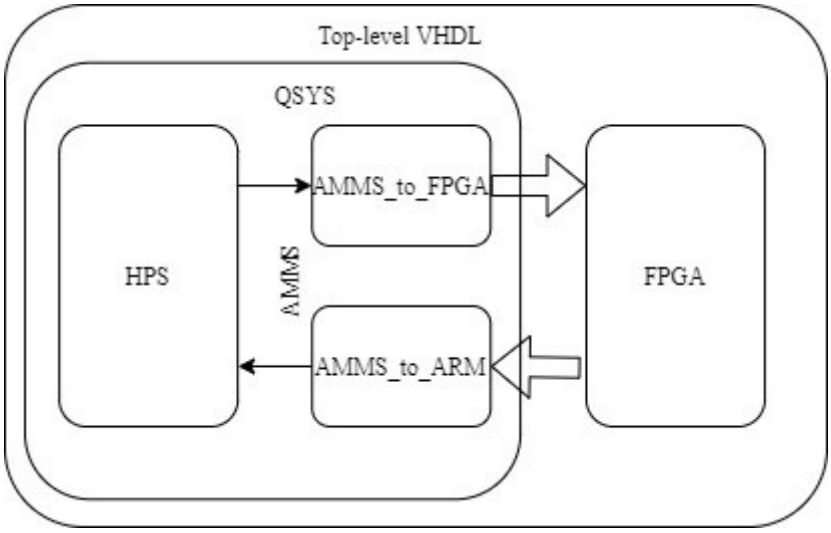
2.6.1 Sillat

Kommunikointi järjestelmien välillä tapahtuu käyttämällä yhtä kolmesta sillasta, joka valitaan käyttötarkoituksen ja tarvittavan tiedonsiirtokapasiteetin mukaan (Kuva 5).



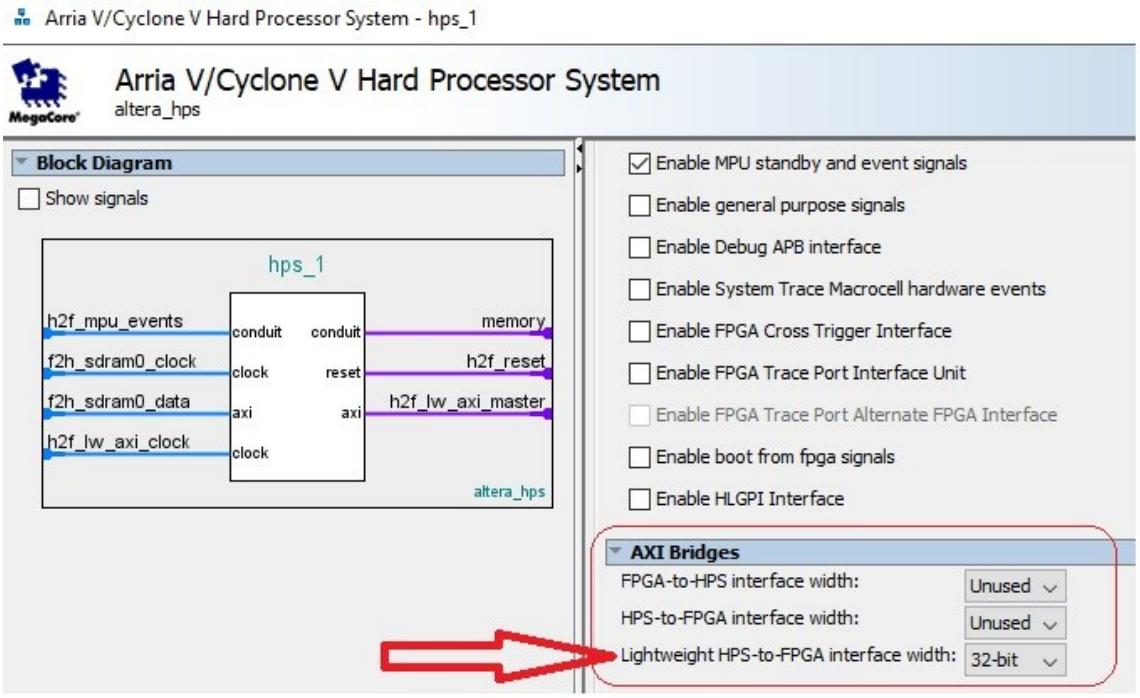
Kuva 5. Lohkokaavio siltojen kommunikoinnista järjestelmän eri osien kanssa. Kaaviossa on merkitty sinisellä sillat, vaaleansinisellä FPGA-osio ja oliivinvihreällä L3-välimuistit prosessorin kanssa kommunikointiin (Intel 2018d).

HPS-to-FPGA-siltaa voidaan käyttää, kun on tarve kontrolloida HPS:llä FPGA:n hallinnassa olevaa muistia. Vastaavasti FPGA-to-HPS-siltaa voidaan käyttää, kun FPGA:lla on tarve päästä HPS:n kontrolloimaan muistiin. Molemmissa näissä silloissa voidaan käyttää 32-, 64-, tai 128-bittistä tietoliikennettä. Kolmas silta on kevytversio ensimmäisestä eli Lightweight HPS-to-FPGA-silta. Tämä kevytsilta tukee 32-bittistä tietoliikennettä, ja sillä pääs-tään käsiksi FPGA-puolen portteihin ja rekistereihin. Tällöin ne ovat orja-asemassa ja HPS on isäntäasemassa (Kuva 6).



Kuva 6. AMMS:in toimintaperiaate ohjelmointiympäristössä. Kuvassa merkitty QSYS, joka rakentaa automaattisesti logiikkayhteydet sillasta prosessoriin (AMMS_to_FPGA) sekä sillasta FPGA:han (AMMS_to_FPGA).

Kevytsiltaa voidaan käyttää, kun ei ole tarvetta suurelle ja nopealle tietoliikenteelle eikä FPGA:n muistiin tarvitse päästä käsiksi. (Intel 2018d.) Tämän vuoksi tähän diplomityöhön on valittu kyseinen silta. Kuva 7 esittää käytettävän sillan valitsemisen Quartus-ohjelmassa.



Kuva 7. Lightweight HPS-to-FPGA -sillan valitseminen Quartus-ohjelmassa. Kyseissä kohdassa voidaan valita myös FPGA-to-HPS tai HPS-to FPGA-silta.

Taulukko 1 esittää käytettävissä olevien siltojen tärkeimmät ominaisuudet.

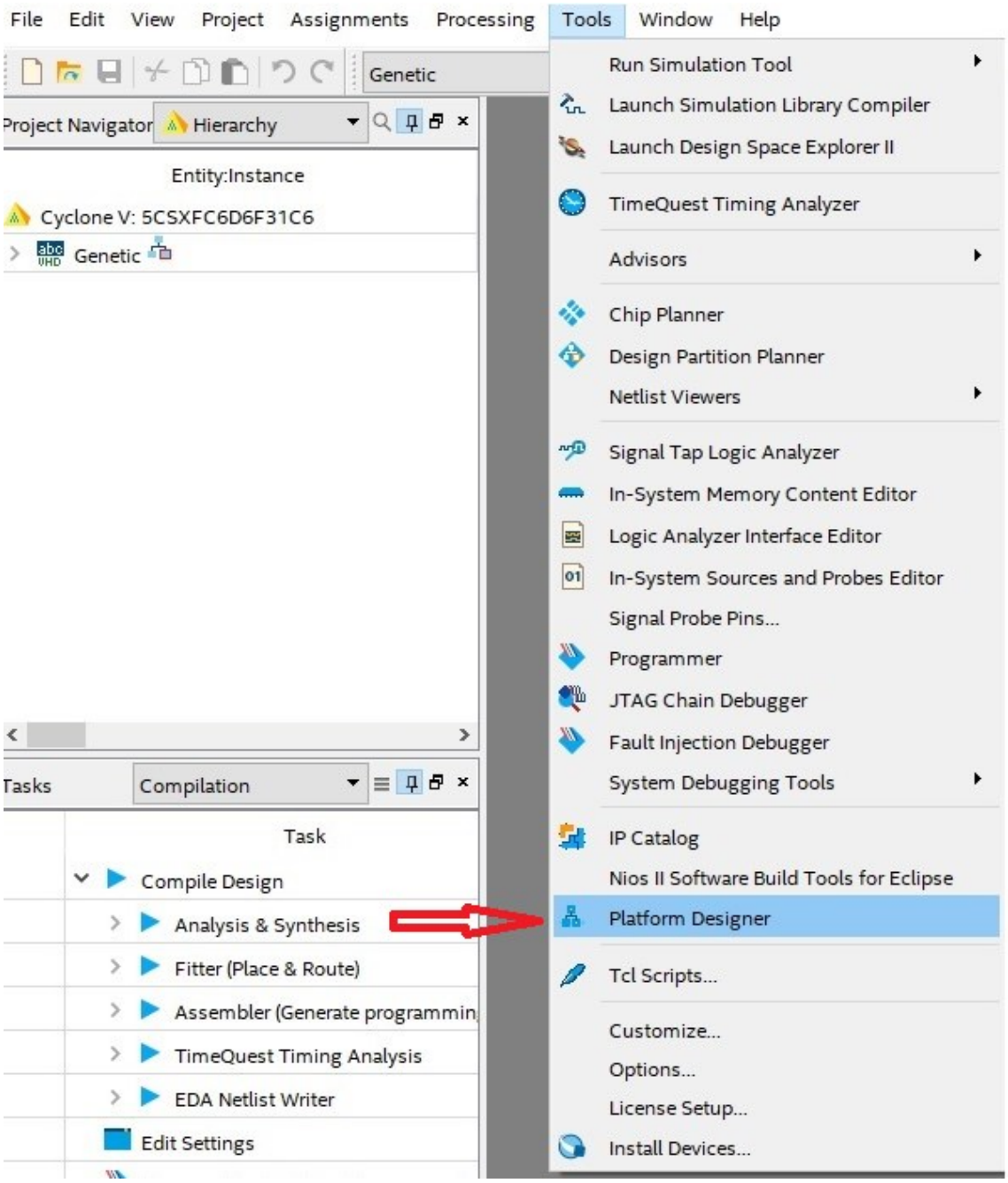
Taulukko 1. AMMS-sillat ja niidet ominaisuudet.

Silta	Bittisyys	Isäntä
Lightweight HPS-to-FPGA	32	HPS
HPS-to-FPGA	32/64/128	HPS
FPGA-to-HPS	32/64/128	FPGA

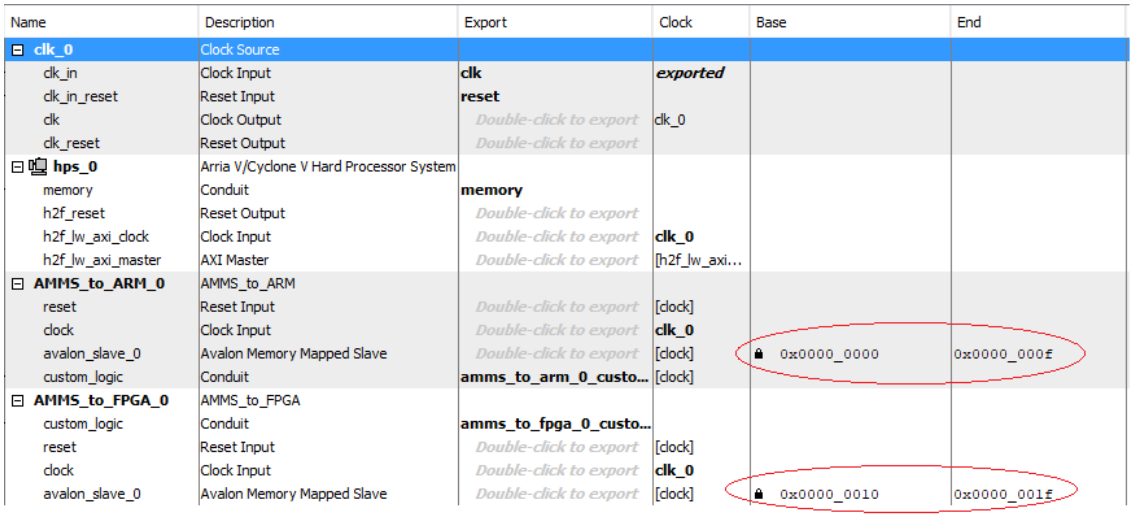
2.6.2 Muistipaikat

Omaan tarpeeseen soveltuvan sillan valitsemisen jälkeen on valittava sopivat muistiosoitteet sekä HPS:n että FPGA:n puolelta. Tarvittava muistipaikkojen määrä riippuu siitä, kuinka suurta määrää dataa järjestelmien välillä tarvitsee siirtää. Jokaista käytettävää FPGA:n muistiosoitetta vastaan on oltava vähintään yhtä monta osoitetta HPS:n puolelta sekä toisinpäin. Esimerkiksi mikäli HPS:n puolelta valitaan muistiosoitteet väliltä 0x0000_0000–0x0000_000f, voidaan vastaavasti FPGA:n puolelta valita muistiosoitteet

väliltä 0x0000_0010–0x_0000_001f. Mikäli lähettävän järjestelmän puolella on enemmän varattuja muistiosoitteita kuin vastaanottavan puolella, osa tiedosta jää siirtymättä. (Intel 2018c.) Kuva 8 ja 9 esittävät muistiosoitteiden valitsemisen Quartuksen Platform Designerissa.



Kuva 8. Platform Designer Quartus-ohjelmassa.



Name	Description	Export	Clock	Base	End
clk_0	Clock Source				
clk_in	Clock Input	**clk**	***exported***		
clk_in_reset	Reset Input	**reset**			
clk	Clock Output	*Double-click to export*	clk_0		
clk_reset	Reset Output	*Double-click to export*			
hps_0	Arria V/Cyclone V Hard Processor System				
memory	Conduit	**memory**			
h2f_reset	Reset Output	*Double-click to export*			
h2f_lw_axi_clock	Clock Input	*Double-click to export*	**clk_0**		
h2f_lw_axi_master	AXI Master	*Double-click to export*	[h2f_lw_axi...		
AMMS_to_ARM_0	AMMS_to_ARM				
reset	Reset Input	*Double-click to export*	[clock]		
clock	Clock Input	*Double-click to export*	**clk_0**		
avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0000	0x0000_000f
custom_logic	Conduit	**amms_to_arm_0_custo...**	[clock]		
AMMS_to_FPGA_0	AMMS_to_FPGA				
custom_logic	Conduit	**amms_to_fpga_0_custo...**			
reset	Reset Input	*Double-click to export*	[clock]		
clock	Clock Input	*Double-click to export*	**clk_0**		
avalon_slave_0	Avalon Memory Mapped Slave	*Double-click to export*	[clock]	0x0000_0010	0x0000_001f

Kuva 9. Muistiosoitteiden valinta HPS:lle ja FPGA:lle Quartus-ohjelman Platform Designerissa.

3 Ohjelmistotestaus

Ohjelmistotestaus on prosessi, jossa varmistetaan, että ohjelma tai ohjelmisto toimii, kuten sen pitää. Testauksen tarkoituksena on siis eri menetelmien avulla joko löytää mahdolliset virheet tai muutoin varmistettava ohjelman toimivuus. Testausmenetelmät voidaan jakaa karkeasti kahteen tyyppiin: staattiseen ja dynaamiseen testaamiseen. Staattisessa testauksessa ohjelmaa ei varsinaisesti käytetä, vaan siinä voidaan tutkia esimerkiksi koodin oikeellisuutta ja rakennetta. Dynaamisessa testauksessa ohjelmaa käytetään ja siihen syötettyjä sekä siitä saatuja arvoja tutkitaan. Dynaamisen alueen testauksesta tässä esitellään lyhyesti kaksi yleistä testaustapaa: lasilaatikkotestaus ja musta laatikko -testaus. (Hetzel 1988: 3–5.)

3.1 Musta laatikko -testaus

Musta laatikko -testaus on testimenetelmä, jossa testattavan ohjelman sisäistä toimintaa ja rakennetta ei tunneta. Ohjelmalle annetaan arvoja ja tutkitaan sen käyttäytymistä ja palautettuja arvoja (Kuva 10). Koska ohjelman rakennetta ja tapahtumia ei tunneta, on testaajan vain varmistettava, että annetut arvot ovat oikeita ja että ohjelma palauttaa odotetut arvot. Jos esimerkiksi ohjelman on annetulla arvolla ’1’ palautettava sama arvo ’1’, ja ohjelma palauttaakin arvon ’0’, testaaja tietää, ettei ohjelma toimi oikein. Testaaja ei kuitenkaan näe, miksi ohjelma ei toimi oikein. (Patton 2006: 55.)

Kuva 10. Musta laatikko -testaus, jossa ohjelman rakennetta ei nähdä.

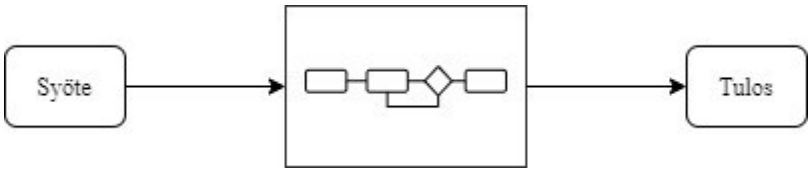
Esimerkiksi AND-veräjää (A AND B) voidaan testata siten, että ohjelmaan, joka kuvaa AND-veräjää syötetään kaikki A:n ja B:n arvot. Totuustaulukosta (Taulukko 2) tiedetään, mitä ohjelman tulisi palauttaa (tummennettuna).

Taulukko 2. AND-veräjän totuustaulukko, jossa tummennettuna odotettu ’1’

A	B	AND
0	0	0
0	1	0
1	0	0
1	**1**	**1**

3.2 Lasilaatikkotestaus

Lasilaatikkotestaus eroaa musta laatikko -testauksesta siten, että siinä testaaja näkee ohjelman rakenteen (Kuva 11). Kuten musta laatikko -testauksessa, voidaan myös lasilaatikkotestauksessa ohjelmaa testata syöttämällä siihen arvoja ja tarkastelemalla tuloksia. Testaus kuitenkin täydentyy, sillä mahdollisen virhetilanteen esiintyessä, testaaja voi paikallistaa virheen kooditasolla asti. Tämä vaatiikin usein testaajalta ohjelmointitaitoa sekä järjestelmän hyvää tuntemusta. Lasilaatikkotestausta käytetäänkin usein osana järjestelmän rakentamista. Ohjelmoija voi esimerkiksi tietyn rakenteen ohjelmoimisen jälkeen testata sen niin, että mahdolliset virheet havaitaan välittömästi jo rakentamisvaiheessa. (Kaner 1999: 41–42.)



Kuva 11. Lasilaatikkotestaus, jossa ohjelman rakenne nähdään.

3.3 Testauksen kattavuus

Testauksen suunnittelussa yksi tärkein valittava asia on testauksen kattavuus. Täydellisellä testaamisella tarkoitetaan sitä, että ohjelmaa testataan kaikilla mahdollisilla arvoilla ja siitä tutkitaan kaikki mahdolliset polut ja tilanteet. Tämä on kuitenkin mahdotonta, sillä syötettäviä arvoja voi olla jopa ääretön määrä ja tutkittavia polkuja erittäin paljon.

Lisäksi mahdollisten syötettävien arvojen lisäksi täytyisi testata myös mahdottomat syötettävät arvot. Seuraavaksi esitellyt kattavuudet kuuluvat kaikki lasilaatikkotestaukseen. (Kaner 1999: 17–19.)

3.3.1 Polkukattavuus

Polkukattavuudessa käydään läpi ohjelman jokainen mahdollinen polku alkulauseesta loppulauseeseen. Lisäksi voidaan olettaa, että ohjelman suorittamisen keskeyttäminen ennen loppulausetta on myös yksi polku. Erilaisia polkuja muodostuu myös esimerkiksi erilaisista silmukoista, jotka voivat muodostaa erilaisia päällekkäisiä polkuja. Täydellisen polkukattavuuden saavuttaminen voi siis olla mahdotonta, koska hiemankaan monimutkaisemmasta ohjelmasta muodostuu lukematon määrä erilaisia polkuja. (Kaner 1999: 18, 43.)

3.3.2 Lausekattavuus

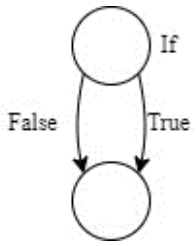
Kattavuuskriteereistä heikoin on lausekattavuus. Lausekattavuus saavutetaan, kun ohjelmasta suoritetaan kerran jokainen ohjelmarivi. Tämä saattaa kuulostaa hyvältä, mutta se jättää huomioimatta esimerkiksi erilaisten ehtolausekkeiden tuottamat vaihtoehdot. (Kaner 1999: 43.) Seuraavassa C-kielisessä esimerkissä esitellään lyhyesti if-lausekkeen läpikäynti lausekattavuudella.

```
int A = 1;

if (A=1);
{
     A=A+1;
}
```

Mikäli ohjelma suoritetaan arvolla A=1, saavutetaan täydellinen lausekattavuus. Tässä tapauksessa huomioimatta jää kuitenkin, mikäli ohjelma suoritetaan esimerkiksi arvolla A=0.

3.3.3 Haara- ja ehtokattavuus

Haarakattavuus on nimensä mukaisesti kriteeri, joka täyttyy, kun ohjelmasta käydään jokainen haaran reuna läpi (Kuva 12). Mikäli jokainen haara käydään läpi, täyttyy myös lausekattavuuden kriteeri.



Kuva 12. Kontrollivuograafi if-lausekkeesta.

Ehtokattavuus on vielä kattavampi. Joissain mahdollisissa tapauksissa kaikki käydyt haarat eivät vielä täytä ehtokattavuutta. Ehtokattavuudessa käydään jokaisen ehtolausekkeen kaikki mahdolliset vaihtoehdot niin, että ne evaluoituvat todeksi sekä epätodeksi. (Kaner 1999: 43–44; Patton 2006: 119–120.) Seuraavassa C-kielisessä esimerkissä esitellään yksi mahdollinen vaihtoehto, jossa haarakattavuus ei käy kaikkia mahdollisia vaihtoehtoja läpi.

```
int A=1;
int B=2;

if (A=1) and (B=2)
{
     A=A+1;
}

else
{
     B=B+1;
}
```

Kuten huomataan, edellisestä ohjelmasta seuraisi samanlainen kontrollivuograafi kuin aiemmassa kuvassa. Haaroja tulisi yhtä monta, mutta kaikkia if-lausekkeen vaihtoehtoja ei käytäisi läpi. Tässä tapauksessa haarakattavuuteen riittäisi vaihtoehdot A=1 ja B=2 sekä

A=0 ja B=2. Tällöin kuitenkin ensimmäinen ehto on kerran tosi ja kerran epätosi, mutta toinen ehto ei koskaan ole epätosi.

4 Toteutus

Tässä osiossa käydään läpi järjestelmän suunnittelu sekä sen toiminta. Lisäksi tarkastellaan geneettisen algoritmin rakennetta, kuten turnajaismenetelmää sekä hyvyysfunktiota.

4.1 Suunnittelu

Järjestelmän suunnittelun lähtökohtana oli valita sopiva testausmenetelmä sekä testauksen kattavuus. Geneettisen algoritmin näkökulmasta kattavuus voi olla mikä tahansa, mistä saa laskettua arvon hyvyysfunktiolle. Käytettävän laitteiston näkökulmasta menetelmä voi olla sellainen, jossa arvoja ja ratkaisuja voidaan helposti välittää HPS:ltä FPGA:lle ja toisinpäin. Tässä työssä käytetään lasilaatikkotestausta, ja kattavuudeksi valittiin haarakattavuusmenetelmä. Se on riittävän kattava toisin kuin esimerkiksi lausekattavuus. Lisäksi testauksen etenemistä on helppo seurata testattavaan ohjelmaan sijoitetulla seurantamenetelmällä. Toisaalta esimerkiksi polkukattavuus olisi vaikeasti seurattava, mikäli testattavassa ohjelmassa on runsaasti silmukoita.

4.1.1 Järjestelmän vaatimukset ja rajoitteet

Järjestelmän vaatimuksena on siis kyetä tuottamaan ratkaisuehdotus ja tämän jälkeen välittämään se testattavalle ohjelmalle. Ohjelma on suoritettava ratkaisuehdotuksen arvoilla ja sen jälkeen tulos on kyettävä lukemaan. Tämän jälkeen ohjelman on tulkittava, onko ratkaisuehdotus riittävä siten, että kaikki haarat on käyty läpi vai tarvitaanko lisää ratkaisuehdotuksia.

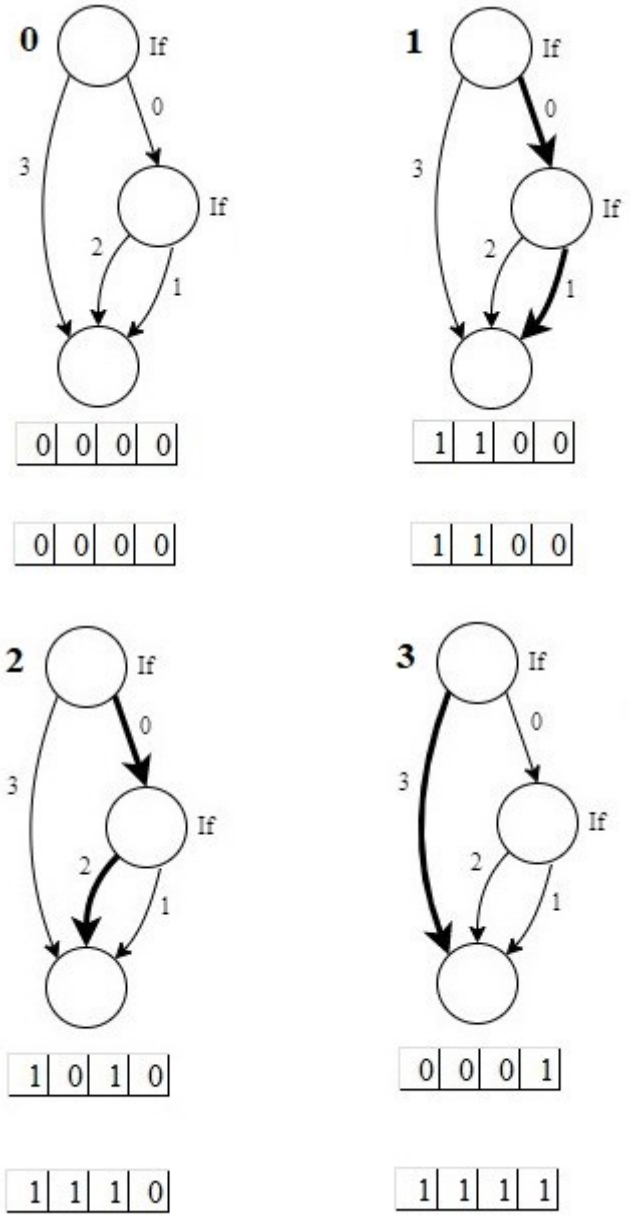
Erityisenä rajoitteena on kommunikointi HPS:n ja FPGA:n välillä. Käytettäviä yhtäaikaisia muistipaikkoja datan liikkumiseen on rajoitettu määrä. Tämä voi rajoittaa datan liikkumista kerralla varsinkin, mikäli testattavassa ohjelmassa on suuri määrä haaroja tai kromosomin pituus on suuri. Tämän vuoksi dataa voidaan joutua lähettämään vaiheittain.

4.1.2 Haarat ja kontrollivuograafi

Esitellään testaamisen lähtökohtana läpikäytävien haarojen muodostamista esimerkkitapauksen VHDL-kielisestä ohjelmasta, missä `A`, `B`, ja `C` ovat muuttujia ja `br()` haaravektori:

```
begin
  K <= 0;
    if A = 1 then
      br(0)=1;
      if B = 1 then
        K <= 1;
        br(1)=1;
      else
        br(2)=1;
      end if;
    else
      br(3)=1;
    end if;
end process;
```

Testattavasta VHDL-kielisestä ohjelmasta muodostetaan kontrollivuograafi (CFG). Tällä graafilla saadaan selkeä kuva läpikäytävien haarojen määrästä sekä niiden sijainnista. Tämän jälkeen kaavion haarat numeroidaan. (Kuva 13). Tällä tavoin saadaan muodostettua haaravektori, joka on lisätty testattavaan ohjelmaan.



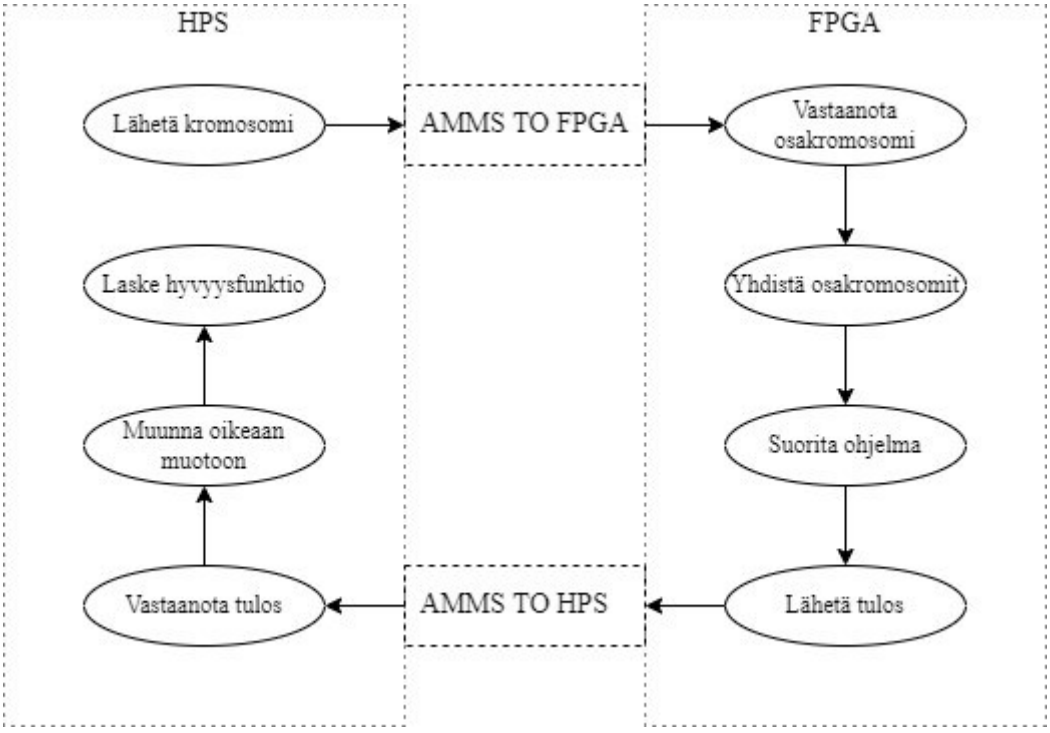
Kuva 13. Esimerkkitapauksen kontrollivuograafi, jossa kaikki haarat on käyty vaiheittain läpi, kunnes haaravektorin arvo on {1}. Vaiheissa 0–4 ylempi vektori on tummennetun haaran vektori, ja alempi vektori on lopullinen haaravektori.

Kuva 13 osoittaa, kuinka haarat käydään vaihe vaiheelta läpi. Alussa (kohta 0) on ilmaistu läpikäytävä graafi ja haaravektorin alkutila. Seuraavassa vaiheessa (kohta 1) käydään läpi ensimmäinen polku, joka sisältää haarat ’0’ ja ’1’. Uutta haaravektoria verrataan aiempaan voimassa olevaan vektoriin (kohdan 0 alempi vektori). Mikäli uudessa vektorissa on uusia haaroja verrattuna aiempaan haaravektoriin, ne lisätään siihen. Toiminto

toistetaan kohdissa 3 ja 4, kunnes haaravektorin arvo on {1}. Kun kaikki haarat on käyty läpi, ohjelma ilmoittaa tulokset.

4.2 Järjestelmän kommunikointi

Kuva 14 esittää testaamisen kommunikoinnin järjestelmän eri osien välillä. HPS-puolelta lähetetään valmis geneettisen algoritmin luoma kromosomi FPGA:lle, joka suorittaa testattavan ohjelman annetuilla arvoilla. Tämän jälkeen testauksen tulos lähetetään takaisin HPS-puolelle geneettiselle algoritmille arvioitavaksi.

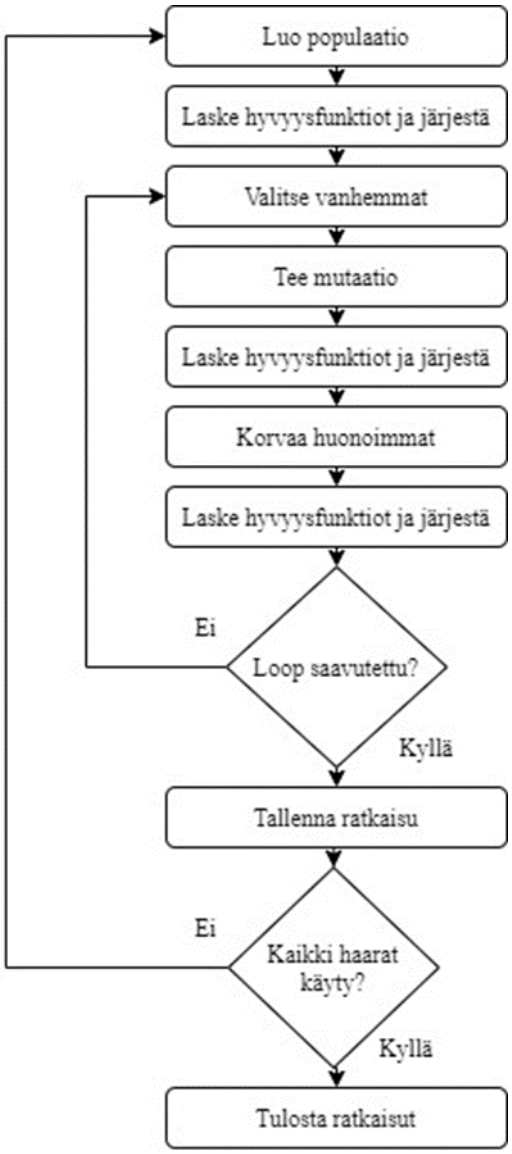


Kuva 14. Järjestelmän toiminta eri osajärjestelmien välillä.

Kun muistipaikkoja on kerrallaan käytössä kahdeksan kappaletta, voidaan FPGA:lle lähettää 8 bitin pituinen vektori. Pidemmät vektorit pilkotaan osiin ja yhdistetään jälleen FPGA:n puolella.

4.3 Geneettisen algoritmin soveltaminen

Geneettisessä algoritmissa luodaan aluksi satunnainen aloituspopulaatio, jonka jälkeen valitaan populaatiosta vanhemmat turnajaismenetelmällä. Turnajaismenetelmän jälkeen vanhemmille tehdään risteytys, jonka jälkeen koko populaatioon suoritetaan mutaatio (Kuva 15).



Kuva 15. Vuokaavio työssä käytetyn geneettisen algoritmin toiminnasta, missä *loop* määrittää, kuinka monta kertaa operaatio suoritetaan.

4.3.1 Populaatio

Populaatio P muodostuu tietystä määrästä kromosomeja. Kromosomin pituus määritellään tässä työssä sen mukaan, millaisella vektorilla testattava ohjelma on käytävä läpi. Esimerkiksi ensimmäisessä testiohjelmassa tarvitaan vektori, joka on neljä bittiä pitkä, joten kromosomin pituus on oltava neljä. Populaation koko eli kromosomien määrä puolestaan voi vaihdella sen mukaan, millaisia tuloksia kullakin populaation koolla saadaan.

4.3.2 Vanhempien valinta ja mutaatio

Vanhempien valinnassa käytetään turnajaismenetelmää, jossa populaatiosta valitaan satunnaisesti tietty määrä kromosomeja, joiden hyvyysarvoa vertaillaan keskenään. Tästä joukosta parhaimman hyvyysarvon omaava kromosomi voittaa ja se valitaan ensimmäiseksi vanhemmaksi. Toinen vanhempi valitaan samalla menetelmällä ja valitut vanhemmat risteytetään keskenään. Tällaisessa valinnassa populaation diversiteetti pysyy laajana, mutta heikomman aineksen mukaan ottaminen saattaa hidastaa globaalin maksimin löytämistä. (Rawlins 1991: 78–82.) Tässä diplomityössä vertailtavien kromosomien määräksi on valittu neljäsosa populaation koosta.

Risteyttämisen jälkeen koko populaatioon suoritetaan mutaatio. Jokaisesta kromosomista valitaan satunnaisesti yksi bitti, joka muutetaan satunnaiseen sallittuun arvoon. Sallittu arvo määritellään aina tapauskohtaisesti. Mutaatiota ei kuitenkaan suoriteta eliittikromosomiin.

4.3.3 Hyvyysfunktio

Hyvyysfunktio f määrittelee kunkin kromosomin soveltuvuuden algoritmin toiminnalle. Hyvyysfunktio määritellään jokaiselle ratkaistavalle ongelmalle omalla tavallaan ja sen määrittely voi vaihdella yksinkertaisesta yhteenlaskusta monimutkaiseen kaavakokonaisuuteen. (Goldberg 1989: 1–11.)

Tässä diplomityössä hyvyysfunktioita on kaksi: ensisijainen ja toissijainen. Ensisijaisen hyvyysfunktion tarkoituksena on järjestää kromosomit järjestykseen sen mukaan, kuinka monta uutta haaraa kukin kromosomi kykenee käymään läpi. Mitä useamman uuden haaran kromosomi käy läpi, sitä parempi se on (kaava 1):

$$f = \sum_{j=1}^{m} b_j, \text{ kun } b_j > b1_j, \quad (1)$$

missä *f* on hyvyysfunktio, *j* on haaran indeksi, *m* on haarojen lukumäärä, *b1* on käytyjen haarojen vektori ja *b* on geneettisen algoritmin ehdottaman polun haarojen vektori. Taulukossa 2 esitetään esimerkkinä uusien ehdotettujen kromosomien vertailu alkuvektoriin {0 0 0 0}. Taulukossa 3 *aa* on vertailtava uusi kromosomi ja *bb* on uusien haarojen lukumäärä. Voittajakromosomit on lihavoitu.

Taulukko 3. Pieni populaatio, jossa kromosomeja verrataan alkuvektoriin {0 0 0 0}.

aa	*bb*
0 0 0 1	1
0 0 1 0	1
0 0 1 1	**2**
0 0 1 1	**2**
0 1 0 1	**2**

Kuten huomataan, ensisijaisen hyvyysfunktion voittaneita kromosomeja voi olla useampi (Taulukko 2). Tämän vuoksi muodostetaan lisäksi myös toissijainen hyvyysfunktio populaation parhaimmille kromosomeille. Koska for-silmukoiden vuoksi polut saattavat muodostua turhan pitkiksi ja raskaiksi, ja algoritmissa haetaan haarakattavuutta, pyritään polkujen pituus minimoimaan. Toissijainen hyvyysfunktio etsii siis lyhimmät polut, jotka vielä täyttävät ensisijaisen hyvyysfunktion kriteerit:

$$f = \sum_{j=1}^{m} b_j, \quad (2)$$

missä *f* on hyvyysfunktio, *j* on haaran indeksi, *m* on haarojen lukumäärä ja *b* on geneettisen algoritmin ehdottaman polun haarojen vektori. Tässä tapauksessa hyvyysfunktio on kuitenkin päinvastainen eli pienempi luku on parempi arvo.

Kahden hyvyysfunktion tapauksessa on kyse Pareto-optimoinnista, jossa pyritään saamaan paras tulos monella eri kriteerillä huonontamatta muiden kriteerien asemaa (Pardalos 2008: 481.) Toissijainen hyvyysfunktio ei siis vaikuta ensimmäisen hyvyysfunktion asemaan. Taulukossa 2 on esimerkki pienestä populaatiosta, jossa kromosomeja verrataan alkuvektoriin {0 0 1 1}. Taulukossa 4 *aa* on kromosomi, ja *cc* on kuljettujen haarojen määrä. Voittajakromosomi on taulukossa lihavoituna.

Taulukko 4. Pieni populaatio, jossa kromosomeja verrataan alkuvektoriin {0 0 1 1}.

aa	*cc*
0 0 1 1	4
0 0 1 1	3
0 1 0 1	**2**

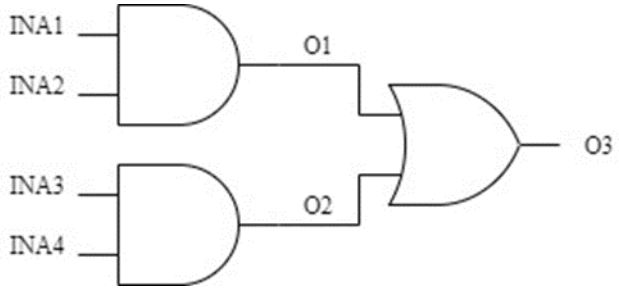
Taulukosta havaitaan, että kromosomi 3 (lihavoitu) käy läpi vähemmän haaroja yhteensä, jolloin se on hyvyysfunktioltaan parempi ja siten tehokkaampi.

5 Tulokset

Tässä kappaleessa käydään läpi viisi testattavaa ohjelmaa. Ohjelmista esitellään niiden lähdekoodit, lähdekoodeihin sijoitetut haarat sekä CFG-kaaviot. Testausta varten jokaisen testattavan ohjelman VHDL-kielinen ohjelma sijoitetaan valmiina olevaan kommunikaatiojärjestelmään omaksi prosessikseen. Tuloksissa esitellään jokaisesta testattavasta ohjelmasta viiden eri testauskerran tulokset. Kaavioissa eritellään geneettisen algoritmin ja satunnaisgeneraattorin tulokset ja niitä vertaillaan keskenään.

5.1 Testiohjelma 1

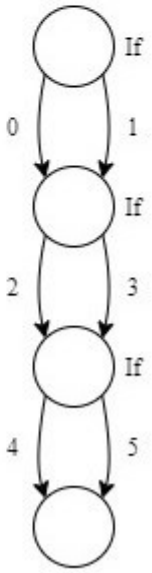
Ensimmäisenä testattavana ohjelmana on muodostettu yksinkertainen logiikkapiiri, jossa on kaksi AND-porttia ja yksi OR-portti (Kuva 16). Logiikkapiirissä on neljä sisääntuloa ja yksi ulostulo.



Kuva 16. Ensimmäisen ohjelman logiikkapiiri, jossa neljä sisääntuloa ja yksi ulostulo.

Seuraavassa esitellään logiikkapiirin VHDL-kielinen ohjelma sekä siitä muodostettu kontrollivuokaavio. Ohjelmasta saadaan hyvin suoraviivainen kokonaisuus, jossa on kuusi läpikäytävää haaraa (Kuva 17):

```
first_case : process(rec_data_logic)

begin

     -- insert logic values
     INA1 <= rec_data_logic(0);
     INA2 <= rec_data_logic(1);
     INA3 <= rec_data_logic(2);
     INA4 <= rec_data_logic(3);

     --insert branches to program
     O1 <= INA1 and INA2;
     O2 <= INA3 and INA4;
     O3 <= O1 or O2;
end process;
```

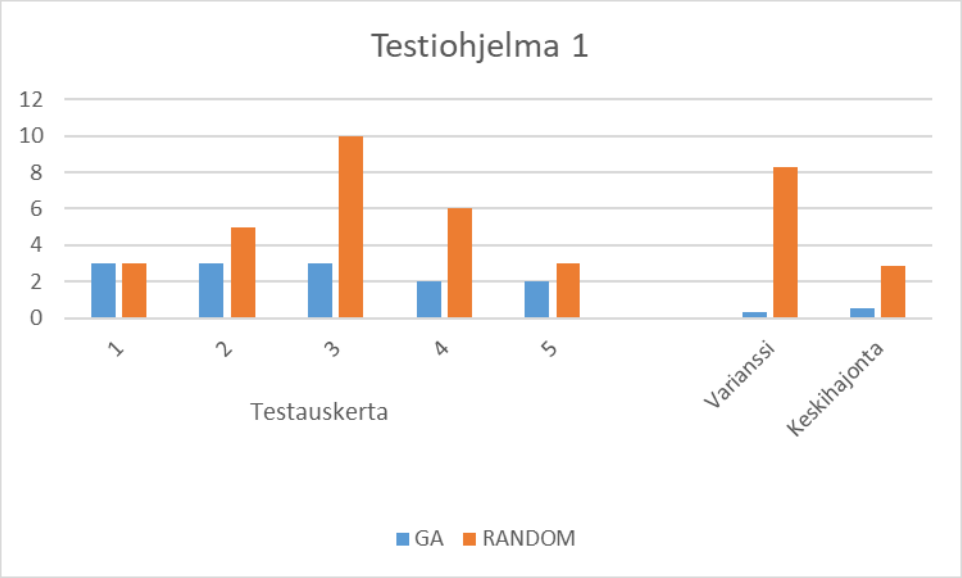


Kuva 17. Ensimmäinen testiohjelma ja kontrollivuokaavio, jossa haarat numeroituna.

Tässä yksinkertaisessa ohjelmassa nähdään, että testitapausten minimäärä, jolla kaikki haarat saadaan käytyä läpi, on kaksi: esimerkiksi haarat ’1’, ’3’, ’5’ ja ’2’, ’4’, ’6’. Sijoitetaan haarat testattavaan ohjelmaan:

```
first_case : process(rec_data_logic)
variable fitness : integer range 0 to 100000;

begin
--reset fitness and branch table
fitness := 0;
init_loop : for i in 0 to 5 loop
     br_1(i) <= 0;
end loop;
-- insert logic values
INA1 <= rec_data_logic(0);
INA2 <= rec_data_logic(1);
INA3 <= rec_data_logic(2);
INA4 <= rec_data_logic(3);
--insert branches to program
O1 <= INA1 and INA2;
if O1 = '1' then
     br_1(0) <= 1;
else
     br_1(1) <= 1;
end if;
O2 <= INA3 and INA4;
if O2 = '1' then
     br_1(2) <= 1;
else
     br_1(3) <= 1;
end if;
O3 <= O1 or O2;
if O3 = '1' then
     br_1(4) <= 1;
else
     br_1(5) <= 1;
end if;

end process;
```

Kuva 18 esittää viiden testikerran tulokset:

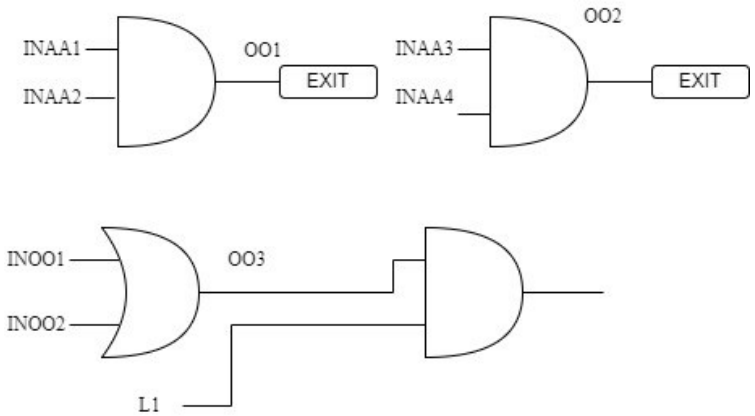


Kuva 18. Ensimmäisen testiohjelman tulokset, jossa vaaka-akselilla testauskerta sekä varianssi ja keskihajonta, ja pystyakselilla ratkaisujen määrä.

Kuten kuvasta 18 huomataan, geneettinen algoritmi ei aina löydä minimimäärää testitapauksia, mutta kuitenkin vaihtelee kahden ja kolmen välillä. Tämä johtunee lyhyestä kromosomin pituudesta, jolloin geneettinen algoritmi voi juuttua lokaaliin maksimiin, ja sen hyödyt jäävät vähäisiksi. Lokaalin maksimiin juuttumista voi yrittää vähentää esimerkiksi lisäämällä mutaatiota. Satunnaisgeneraattorin löytämät testitapaukset kuitenkin vaihtelevat rajusti.

5.2 Testiohjelma 2

Toisena testattavana ohjelmana on piiri, jossa on kaksi AND-porttia ja yksi OR-portti siten, että ohjelma päättyy aina, kun minkä tahansa portin tila on ’1’ (Kuva 19).

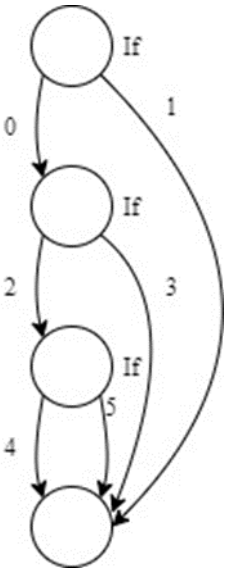


Kuva 19. Testiohjelman 2 logiikkapiiri.

Seuraavassa esitellään testattavan ohjelman VHDL-kielinen koodi.

```
second_case : process(rec_data_logic2)
variable a : integer range 0 to 1;

begin
--while loop variable
a := 0;
-- insert logic values
INAA1 <= rec_data_logic2(0);
INAA2 <= rec_data_logic2(1);
INAA3 <= rec_data_logic2(2);
INAA4 <= rec_data_logic2(3);
INOO1 <= rec_data_logic2(4);
INOO2 <= rec_data_logic2(5);

--loop to make it possible to exit
while ( a = 0) loop
     OO1 <= INAA1 and INAA2;
     if OO1 = '1' then
               exit;
     end if;

     OO2 <= INAA3 and INAA4;
     if OO2 = '1' then
          exit;
     end if;

     OO3 <= INOO1 or INOO2;
     if OO3 = '1' then
          exit;
     end if;
     a := 1;
end loop;
end process;
```

Kuva 20 esittää ohjelmasta muodostuvat haarat, jotka tulee käydä läpi.

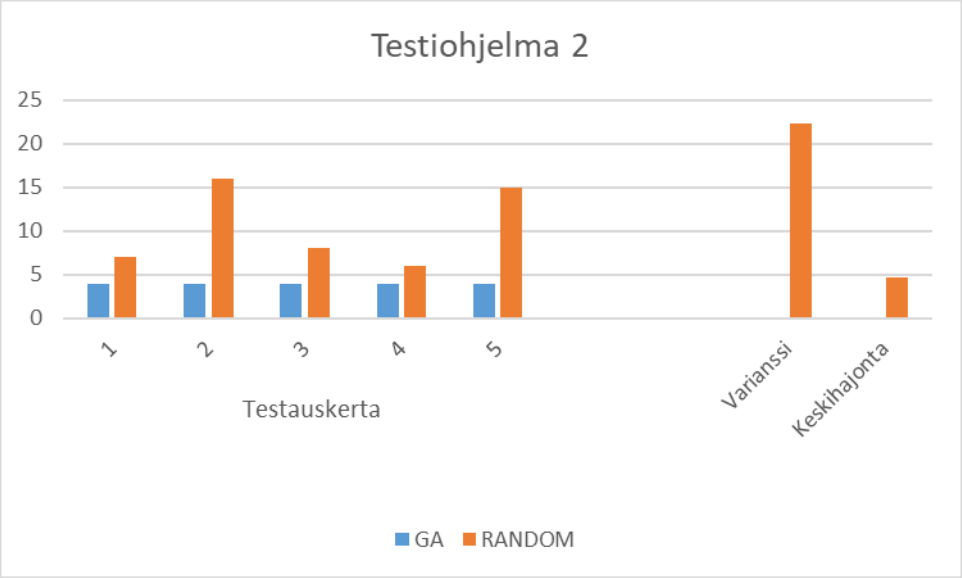


Kuva 20. Toisen testitapauksen CFG, jossa haarat numeroituna.

Kuten huomataan, vaikka haarojen lukumäärä pysyy samana ensimmäiseen testitapaukseen verrattuna, ei kaikkia haaroja voida käydä läpi vain kahdella muodostetulla ratkaisulla. Sijoitetaan haarat testattavaan ohjelmaan:

```
second_case : process(rec_data_logic2)
variable a : integer range 0 to 1;

begin

--while loop variable
a := 0;

-- insert logic values
INAA1 <= rec_data_logic2(0);
INAA2 <= rec_data_logic2(1);
INAA3 <= rec_data_logic2(2);
INAA4 <= rec_data_logic2(3);
INOO1 <= rec_data_logic2(4);
INOO2 <= rec_data_logic2(5);

--insert branches to program
while ( a = 0) loop
     OO1 <= INAA1 and INAA2;
     if OO1 = '1' then
               br_2(0) <= 1;
               exit;
     else
          br_2(1) <= 1;
     end if;
     OO2 <= INAA3 and INAA4;
     if OO2 = '1' then
          br_2(2) <= 1;
          exit;
     else
          br_2(3) <= 1;
     end if;
     OO3 <= INOO1 or INOO2;
     if OO3 = '1' then
          br_2(4) <= 1;
          exit;
     else
          br_2(5) <= 1;
     end if;
     a := 1;
end loop;


end process;
```

Kuva 21 esittää viiden testikerran tulokset:



Kuva 21. Toisen testiohjelman tulokset, jossa vaaka-akselilla testauskerta sekä varianssi ja keskihajonta, ja pystyakselilla ratkaisujen määrä.

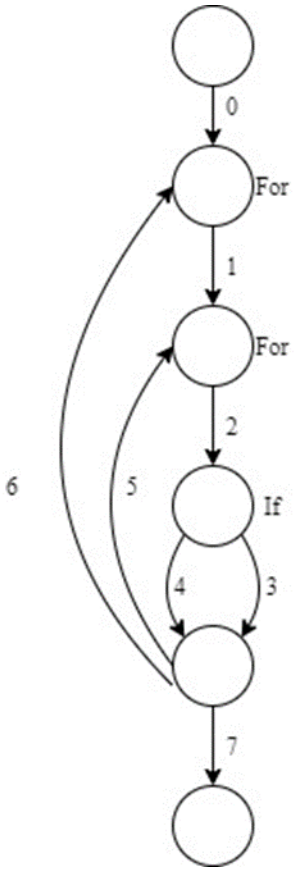
Kuva 21 kaaviosta käy ilmi, että geneettinen algoritmi löytää minimimäärän testiratkaisuja jokaisella testauskerralla. Satunnaisesti luotujen testiratkaisujen määrä puolestaan vaihtelee huomattavasti, aivan kuten ensimmäisessä testitapauksessa. Hajonnassa testitapausten minimimäärän muuttuessa huomataan muutos. Satunnaisgeneraattorin hajonta on melko suurta, kun geneettisellä algoritmilla hajontaa ei ole yhtään.

5.3 Testiohjelma 3

Kolmantena testattavana ohjelmana on lukujen järjestämisalgoritmi eli ”bubble sort”. Algoritmiin lähetetään kahdeksan numeron pituinen vektori, jonka kokonaisluvut ovat tässä tapauksessa 0–100. Algoritmi järjestää numerot suuruusjärjestykseen pienimmästä aloittaen. Seuraavassa esitellään algoritmin koodi:

```
third_case : process(rec_data2)
--variables
variable bubble : bubble_array := (0,0,0,0,0,0,0,0);
variable swap : integer range 0 to 100;
begin
--insert values to variable array
init_loop2 : for i in 0 to 6 loop
     bubble(i) := rec_data1(i);
end loop;

swap := 0;

first_loop : for i in 0 to 7 loop
     second_loop : for j in 0 to 7-i-1 loop
          --check if next number is bigger
          if bubble(j) > bubble(j+1) then
               --swap the numbers
               swap := bubble(j);
               bubble(j) := bubble(j+1);
               bubble(j+1) := swap;
          else
          end if;
     end loop;
end loop;
end process;
```

Muodostetaan koodista kontrollivuokaavio (Kuva 22).

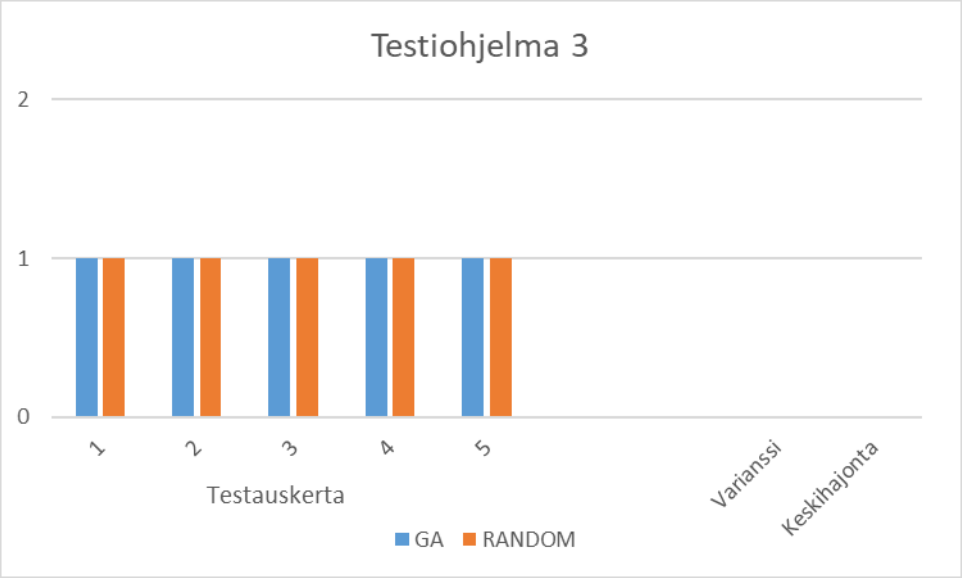


Kuva 22. Kolmannen testattavan ohjelman kontrollivuokaavio.

Kuten kuvasta nähdään, kaikki haarat on mahdollista käydä läpi yhdellä testitapauksella for-silmukoita hyödyntäen. Sijoitetaan haarat koodiin:

```
third_case : process(rec_data2)
variable bubble : bubble_array := (0,0,0,0,0,0,0,0);
variable swap : integer range 0 to 100;
variable fitness : integer range 0 to 100000;
variable flag : integer range 0 to 10;
variable flag2 : integer range 0 to 10;
begin

init_loop2 : for i in 0 to 7 loop
     bubble(i) := rec_data1(i);
end loop;

br_3(0) <= 1;

fitness := 0;
swap := 0;

first_loop : for i in 0 to 7 loop
     br_3(1) <= 1;
     flag := flag+1;
     second_loop : for j in 0 to 7-i-1 loop
          br_3(2) <= 1;
          flag2 := flag2+1;
          --if next number is greater
          if bubble(j) > bubble(j+1) then
               br_3(3) <= 1;
               --swap the numbers
               swap := bubble(j);
               bubble(j) := bubble(j+1);
               bubble(j+1) := swap;
          else
               br_3(4) <= 1;
          end if;
          --if looped back
          if flag2 > 1 then
               br_3(5) <= 1;
          end if;
     end loop;
     flag2 := 0;
     --if looped back
     if flag > 1 then;
          br_3(6) <= 1;
     end if;
end loop;
br_3(7) <= 1;
flag := 0;

end process;
```

Tutkitaan testattavaa ohjelmalla viidellä eri testauskerralla (Kuva 23).



Kuva 23. Toisen testiohjelman tulokset, jossa vaaka-akselilla testauskerta sekä varianssi ja keskihajonta, ja pystyakselilla ratkaisujen määrä.

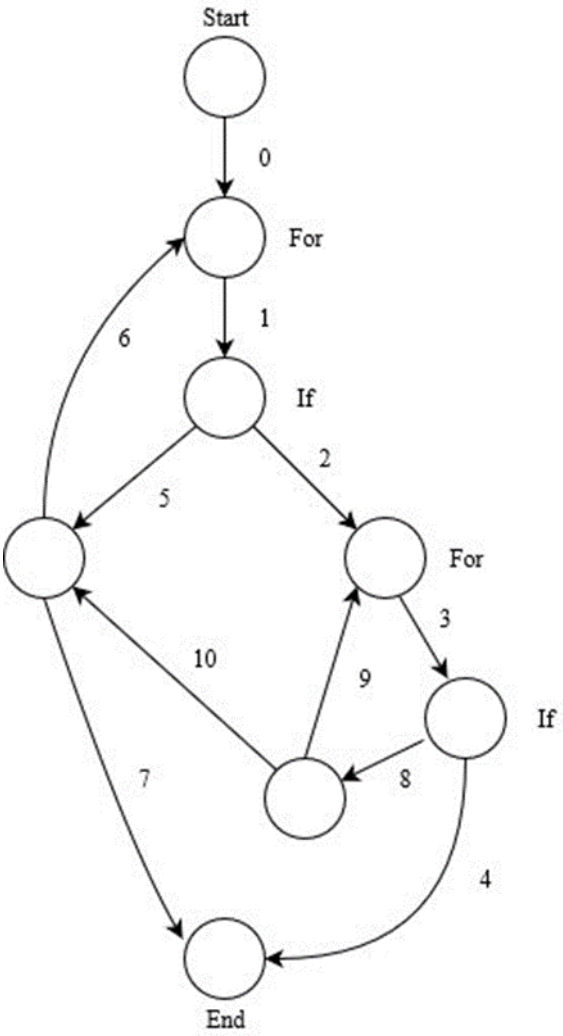
Huomataan, että geneettinen algoritmi ja satunnaisgeneraattori löytävät jokaisella testauskerralla minimimäärän testitapauksia. Tästä voidaan päätellä, että testattavan algoritmin rakenne on sellainen, että lähes jokainen testattava ratkaisu käy läpi kaikki haarat kerralla. Tässä tapauksessa voidaan päätellä, että geneettisestä algoritmista ei ole suurempaa hyötyä kyseisessä tapauksessa valitulla testaustavalla. Myöskään hajontaa ei ole.

5.4 Testiohjelma 4

Neljäntenä testattavana ohjelmana on algoritmi, joka käy läpi annetun totuustaulukon arvot ja tutkii, onko kyseinen ratkaisu sallittu vai kielletty. Tätä testausta varten ohjelma on käännetty C-koodista VHDL-koodiin. Kyseinen ohjelma valittiin siksi, että siinä on for-silmukoita, jotka monimutkaistavat ongelmaa lisäämällä läpikäytäviä haaroja. Seuraavassa koodissa esitellään testattava ohjelma:

```
begin
flag <= 0;
--first loop
first_loop : for i in 0 to 7 loop
     --check if there if is true
     if rec_data(i) = 1 then
          second_loop : for j in i+1 to 7 loop
               --check if allowed
               if rec_data(j) = 1 and rule_data(i, j) = 0
               then
                    --not allowed, flag and stop
                    flag <= 1;
                    exit first_loop;
                    exit second_loop;
               end if;
          end loop;
     end if;
end loop;
end process;
```

Melko pienestä koodin määrästä huolimatta haarojen määrä on jo suurempi kuin edellisessä testattavassa ohjelmassa. Muodostetaan ohjelmasta kontrollivuokaavio (Kuva 24).

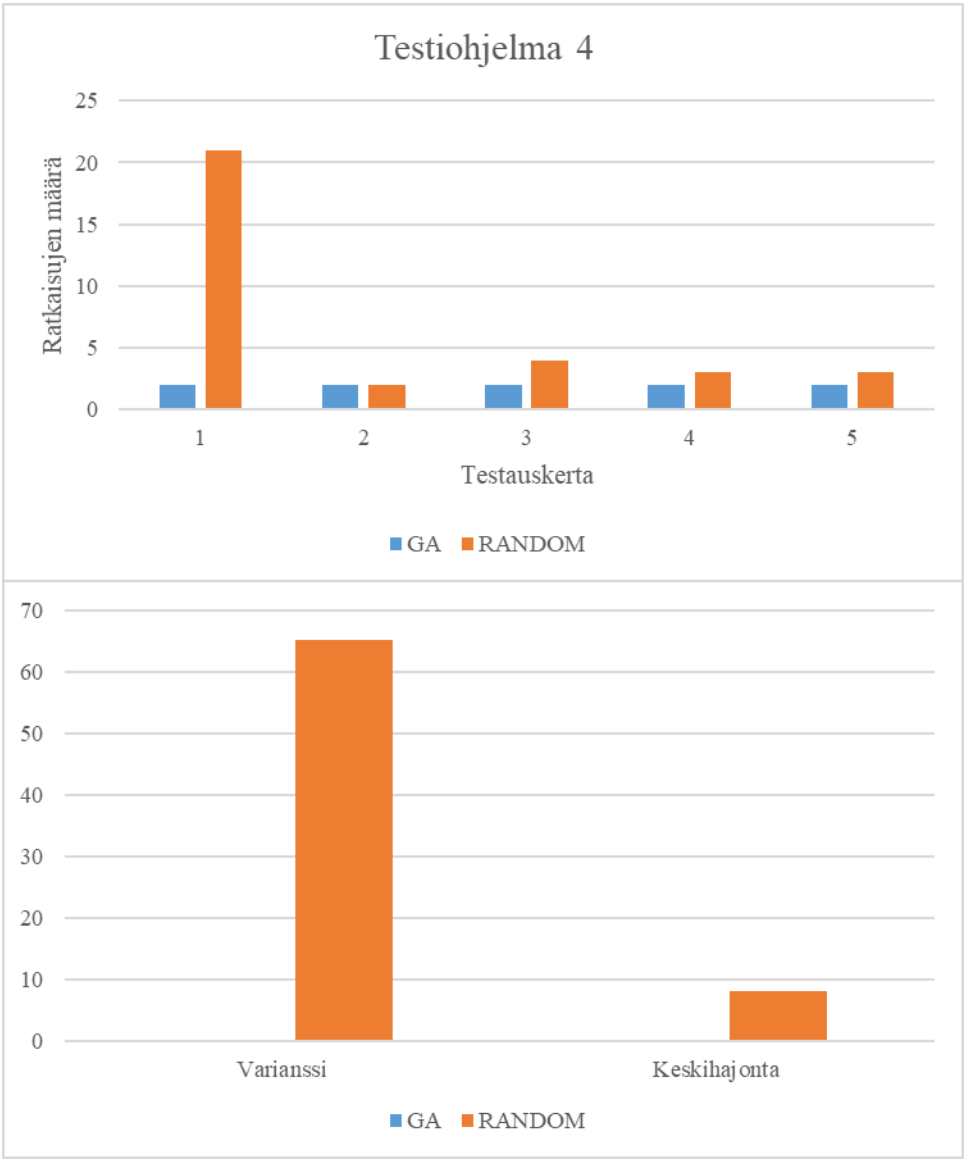


Kuva 24. Neljännen testattavan ohjelman kontrollivuokaavio, jossa näkyy läpikäytävät haarat.

Sijoitetaan koodiin haarojen paikat:

```
begin

br_4(0) <= 1;
--flags for for loops
flag <= 0;
flag2 := 0;
flag3 := 0;
fitness := 0;

first_loop : for i in 0 to 7 loop
     br_4(1) <= 1;
     if rec_data2(i) = 1 then
          br_4(2) <= 1;
          second_loop : for j in i+1 to 7 loop
               br_4(3) <= 1;
               flag3 := flag3+1;
               if rec_data2(j) = 1 and rule_data(i, j) = 0
               then
                    br_4(4) <= 1;
                    flag <= 1;
                    exit first_loop;
                    exit second_loop;
               else
                    br_4(8) <= 1;
               end if;
               --if looped back
               if flag3 > 1 then
                    br_4(9) <= 1;
               end if;
          end loop;
          flag3 := 0;
          br_4(10) <= 1;
     else
          br_4(5) <= 1;
     end if;
     --if looped back
     if flag2 > 1 then
          br_4(6) <= 1;
     end if;
end loop;
flag2 := 0;
if flag = 0 then
     br_4(7) <= 1;
end if;
end process;
```

Aiempien testattavien ohjelmien tapaan esitellään viiden eri testauskerran tulokset (Kuva 25):



Kuva 25. Neljännen testattavan ohjelman tulokset, jossa ylempänä vaaka-akselilla testikerta ja pystyakselilla ratkaisujen määrä, ja alempana varianssi sekä keskihajonta.

Vaikka ohjelmassa on monta testattavaa haaraa silmukoineen, huomataan, että kaikki haarat on mahdollista käydä läpi vain kahdella testitapauksella. Geneettinen algoritmi näyttää jälleen löytävän jokaisella testauskerralla minimäärän tapauksia, joilla kaikki haarat voidaan käydä läpi. Myös satunnaisgeneraattori melko pienen määrän testitapauksia, lukuun ottamatta suurta poikkeamaa ensimmäisellä testauskerralla. Tässä tapauksessa

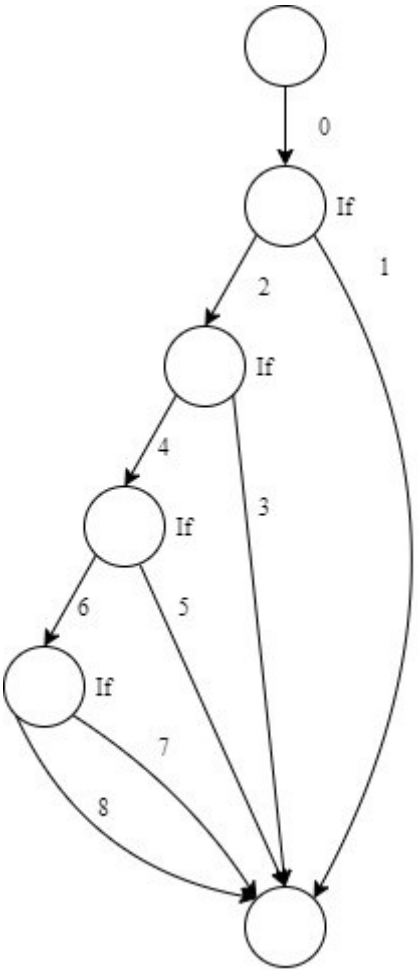
voidaan päätellä, että geneettisestä algoritmista on hyötyä testitapausten etsimisessä. Koska geneettinen algoritmi löysi jokaisella testauskerralla minimimäärän testitapauksia, hajontaa ei ole. Tämä on siten ideaalinen tilanne ja tähän myös pyritään. Satunnaisgeneraattorissa hajonta on jälleen melko suurta.

5.5 Testiohjelma 5

Viidentenä testitapauksena on ohjelma, joka tutkii kuvapikselin annettuja arvoja ja luokittelee niiden perusteella, mihin maastoluokkaan kyseinen pikseli kuuluu. Testattava osuus ohjelmasta on päätöspuualgoritmi. Kyseinen ohjelma on valittu siksi, että se sisältää suuren määrän sisääntuloja, eli kromosomi on mahdollisimman pitkä. Kromosomin pituuden vuoksi sitä ei voi lähettää logiikkapuolelle kerralla, vaan se on pilkottava ja lähetettävä osissa. Seuraavassa koodissa esitellään testattava ohjelma:

```
begin
     if featurebuffer(4) < 2461 then
          class <= 1;
          ready := 1;
     end if;
     if ((featurebuffer(10)) < 4116) and (ready = 0) then
          class <= 2;
          ready := 1;
     end if;
     if((featurebuffer(11)) > 5148) and (ready = 0) then
          class <= 5;
          ready := 1;
     end if;
     if ((featurebuffer(6)) < 8474) and (ready = 0) then
          class <= 3;
          ready := 1;
     elsif (ready = 0) then
          class <= 4;
          ready := 1;
     end if;
     ready := 0;
end process;
```

Muodostetaan koodista kontrollivuokaavio (Kuva 26):

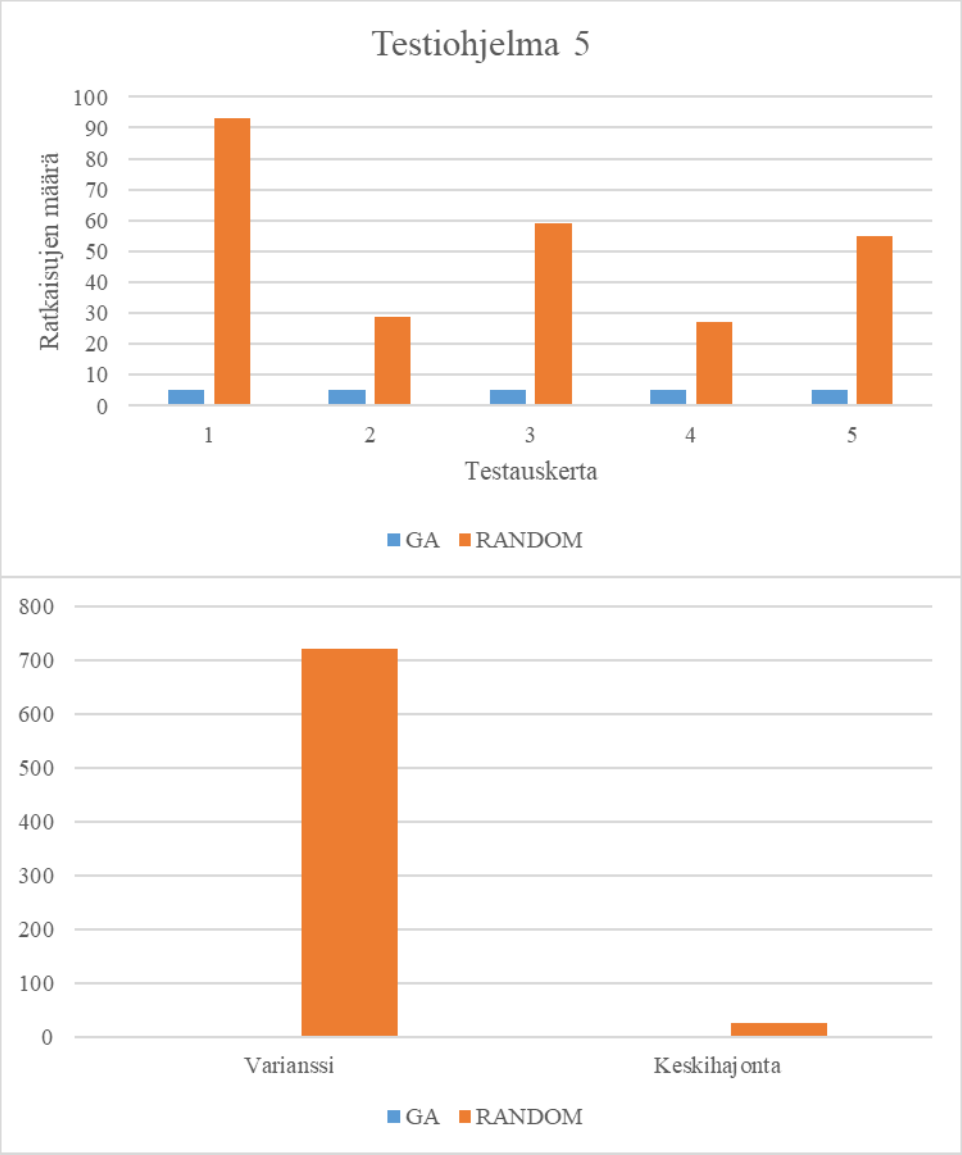


Kuva 26. Viidennen testattavan ohjelman kontrollivuokaavio, jossa näkyy läpikäytävät haarat.

Sijoitetaan koodiin haarat:

```
calculate_class : process(featurebuffer)
 variable ready:     natural := 0;
 begin
      fitness := 0;
      ready := 0;
      br_5(0) <= 1;
      if featurebuffer(4) < 2461 then
           br_5(1) <= 1;
           class <= 1;
           ready := 1;
      else
           br_5(2) <= 1;
      end if;
      if ((featurebuffer(10)) < 4116) and (ready = 0) then

           br_5(3) <= 1;
           class <= 2;
           ready := 1;
      elsif (ready = 0) then
           br_5(4) <= 1;
      end if;
      if((featurebuffer(11)) > 5148) and (ready = 0) then
           br_5(5) <= 1;
           class <= 5;
           ready := 1;
      elsif (ready = 0) then
           br_5(6) <= 1;
      end if;
      if ((featurebuffer(6)) < 8474) and (ready = 0) then
           br_5(7) <= 1;
           class <= 3;
           ready := 1;
      elsif (ready = 0) then
           br_5(8) <= 1;
           class <= 4;
           ready := 1;
      end if;
      ready := 0;
 end process;
```

Esitellään tulokset (Kuva 27):

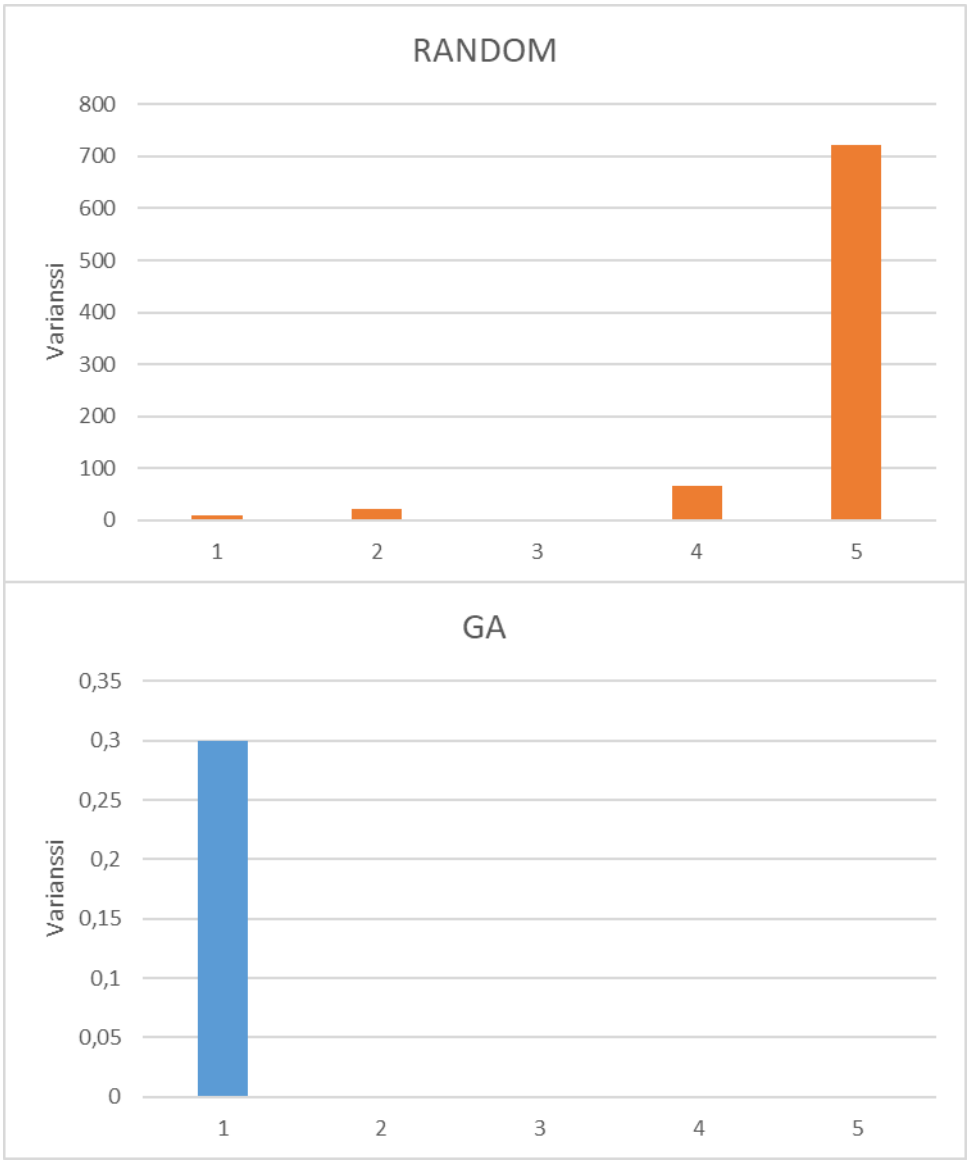


Kuva 27. Viidennen testattavan ohjelman tulokset, jossa ylempänä vaaka-akselilla testikerta ja pystyakselilla ratkaisujen määrä, ja alempana varianssi sekä keskihajonta.

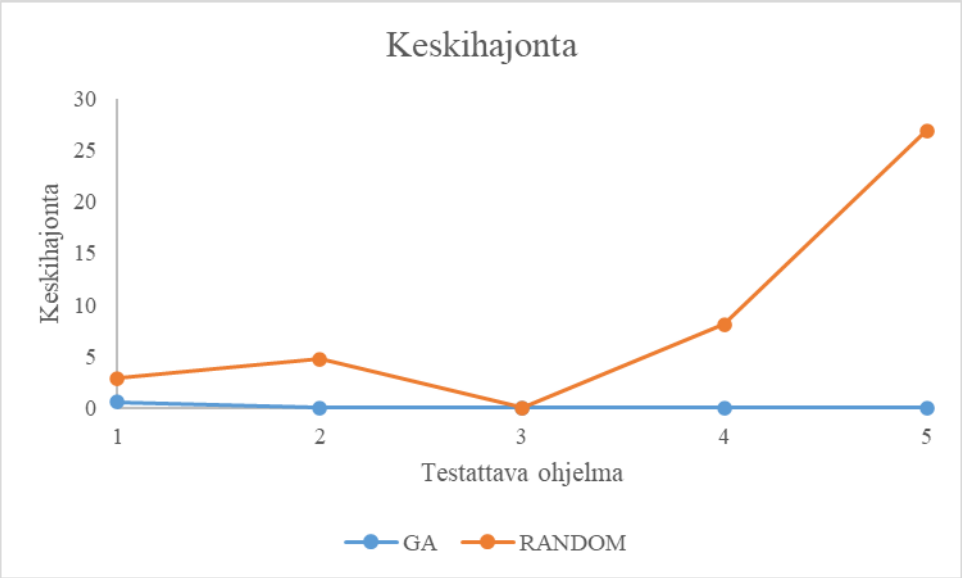
Geneettinen algoritmi löysi jälleen kerran jokaisella testauskerralla minimimäärän testitapauksia. Tämä vahvistaa ajatusta, että mitä pidempi kromosomi on, sitä paremmin geneettinen algoritmi toimii. Varsinkin, jos läpikäytäviä haaroja on runsaasti, on geneettinen algoritmi erittäin tehokas varsinkin satunnaiseen testaamiseen verrattuna. Jälleen, geneettisen algoritmin tapauksessa hajontaa ei ole.

5.6 Yhteenveto

Kaikkia ohjelmia testattiin viisi kertaa ja tulokset kirjattiin ylös. Kuva 28 ja Kuva 29 on esitetty testitapausten löytymisen suhteen geneettisen algoritmin ja satunnaisgeneraattorin varianssi ja keskihajonta.



Kuva 28. Viiden eri testattavan ohjelman varianssi testitapausten löytymisen suhteen. Ylhäällä satunnainen ja alhaalla geneettinen algoritmi.



Kuva 29. Viiden eri testattavan ohjelman keskihajonta testitapausten löytymisen suhteen.

Voidaan todeta, että geneettinen algoritmi löytää minimimäärän testitapauksia erittäin hyvin. Aivan yksinkertaisimmassa eli ensimmäisessä testattavassa ohjelmassa on pientä hajontaa, eli algoritmi joskus juuttuu lokaaliin maksimiin. Tämän oletan johtuvan erittäin lyhyestä kromosomista. Tällöin geneettinen algoritmi ei tuota aina suurta hyötyä paremman ratkaisun etsimiselle. Suurin hyöty geneettisestä algoritmista saadaan, kun kromosomi on tarpeeksi pitkä, ja haaroja on useampi. Neljännen ja viidennen testattavan ohjelman kohdalla voidaankin tämä todeta, sillä satunnaisgeneraattorilla hajonta on huomattavan suurta, kun geneettisen algoritmin kohdalla sitä ei ole. Tämä onkin testausohjelman tavoite.

6 Johtopäätökset

Diplomityön tavoitteena oli rakentaa SoC FPGA:lle geneettisellä algoritmilla toimiva ohjelma, joka löytää minimimäärän testitapauksia VHDL-ohjelman testaamiseen. Ohjelman rakenne tuli olla sellainen, että geneettinen algoritmi sijaitsee laitteen HPS-puolella ja testattava VHDL-ohjelma sijaitsee logiikkapuolella. Tähän liittyen tavoitteena oli myös tutkia SoC FPGA:n soveltuvuutta kyseiseen testaamiseen.

Työtä lähdettiin suunnittelemaan tutkimalla SoC FPGA:n ominaisuuksia ja millaista dataa ja millaisessa muodossa siinä on mahdollista lähettää järjestelmältä toiselle. Valmiiden kommunikaatiokirjastojen avulla HPS-puolelta saatiin lähetettyä geneettisen algoritmin ehdottamia kromosomeja FPGA-puolelle. Kyseiset kromosomit voivat sisältää mitä tahansa kokonaislukuja. Testattavalla ohjelmalla piti olla riittävästi sisääntuloja, jotta kromosomi olisi tarpeeksi pitkä.

Seuraavaksi oli harkittava, millaista kattavuutta ja testaustapaa haluttiin tavoitella. Kattavuuden tuli olla tarpeeksi riittävä ja tuoda testaukseen sopivaa haastetta. Esimerkiksi lausekattavuus olisi ollut helpompi toteuttaa, mutta tätä en kuitenkaan pitänyt riittävänä. Polkukattavuus olisi testattavissa ohjelmissa sijaitsevien silmukoiden vuoksi voinut olla liian monimutkainen. Sopivaksi kattavuusmääritelmäksi jäikin tässä työssä haarakattavuus.

Suurimmiksi ongelmiksi muodostuivat jo läpikäytyjen haarojen seuraaminen ja huomioon ottaminen sekä pitkän kromosomin lähettäminen FPGA:lle. Haarojen seuraaminen onnistui niiden numeroinnilla ja erilliseen haaravektoriin sijoittamisella. Mikäli haaroja kuitenkin muodostuisi erityisen monta, esimerkiksi kymmeniä tai satoja, tällaisen vektorin lähettäminen takaisin geneettiselle algoritmille muodostuisi vaikeaksi muistipaikkojen loppumisen vuoksi. Tämä kuitenkin ratkaistiin muuntamalla bittivektoria eri muotoon lähetystä varten. Pitkä kromosomi puolestaan saatiin lähetettyä FPGA-puolelle lähettämällä se osissa.

Testausohjelmaa voidaan käyttää esimerkiksi testitapausten etsimiseen pienemmissä ohjelmissa sekä suurempien järjestelmien eri osissa. Suurten tai monimutkaisten järjestelmien testauksessa on kuitenkin huomattava, että tämä testausohjelma perustuu siihen, että testattavassa koodissa on valmiina sijoiteltu koodin haarojen paikat. Tällöin testattavaan ohjelmaan tulisi jo ohjelmointivaiheessa tiedostaa ja ohjelmoida kyseiset kohdat tai sijoittaa ne koodiin erillisellä ohjelmalla. Lisäksi on huomioitava, että geneettisen algoritmin kromosomin osat tulee aina olla samaa muotoa kuin testattavan ohjelman sisääntulot. Tämän vuoksi algoritmi joudutaan aina räätälöimään sopivaksi jokaista erillistä testattavaa ohjelmaa varten.

Lähteet

Alander, Jarmo (2014). *An Indexed Bibliography of Genetic Algorithms in Testing* [online]. Vaasan yliopisto [viitattu 16.6.2020]. Saatavissa: <http://www.uva.fi/~TAU/reports/report94-1/gaTESTbib.pdf>

Arm (2007). *Media Alert: Five Billionth ARM Processor for Mobile Devices* [online]. Cambridge, England: Arm Holdings. [viitattu 9.11.2018]. Saatavissa: <https://www.arm.com/about/newsroom/16535.php>

Arm (2018). *Widely Deployed Multicore Processor* [online]. Cambridge, England: Arm Holdings. [viitattu 9.11.2018]. Saatavissa: <https://www.arm.com/products/silicon-ip-cpu/cortex-a/cortex-a9>

Ashenden, Peter J. (2008). *Digital Design. An Embedded Systems Approach Using Verilog*. Burlington, USA: Elsevier Inc. ISBN 978-0-12-369527-7

Chu, Pong P. (2008). *FPGA Prototyping by VHDL Examples*. USA: John Wiley & Sons, Inc. ISBN 978-0-470-18531-5

Clark, Don (2015). *Intel Completes Acquisition of Altera* [online]. New York City: Wall Street Journal. [viitattu 7.11.2018]. Saatavissa: <https://www.wsj.com/articles/intel-completes-acquisition-of-altera-1451338307/>

Grout, Ian (2008). *Digital Systems Design with FPGAs and CPLDs*. Oxford: Elsevier Ltd. ISBN 978-0-7506-8397-5.

Hetzel, Bill. (1988). *The Complete Guide to Software Testing*. New York: John Wiley & Sons, Inc. ISBN 0-471-56567-9

Intel (2014). *What is an SoC FPGA?* [online]. Santa Clara, California: Intel Corporation [viitattu 1.10.2018]. Saatavilla: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ab/ab1_soc_fpga.pdf>

Intel (2018a). *Cyclone V Hard Processor System Technical Reference Manual* [online]. Santa Clara, California: Intel Corporation [viitattu 14.11.2018]. Saatavissa: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_5v4.pdf>

Intel (2018b). *Cyclone V FPGAs* [online]. Santa Clara, California: Intel Corporation [viitattu 12.12.2018]. Saatavissa: <https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-v.html>

Intel (2018c). *Avalon Interface Specification* [online]. Santa Clara, California: Intel Corporation [viitattu 21.11.2018]. Saatavissa: <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf>

Intel (2018d). *Guidelines for Interconnecting the HPS and FPGA* [online]. Santa Clara, California: Intel Corporation [viitattu 22.11.2018]. Saatavissa: <https://www.intel.com/content/altera-www/global/en_us/index/documentation/doq1481305867183/ocx1481303252791/sxr1481303255441.html>

Intel (2018e). *Intel Quartus Prime Software Suite* [online]. Santa Clara, California: Intel Corporation. [viitattu 8.1.2019]. Saatavissa: <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html>

Intel (2019). *Introduction to the Hard Processor system* [online]. Santa Clara, California: Intel Corporation [viitattu 07.01.2019]. Saatavissa: <https://www.intel.com/content/www/us/en/programmable/documentation/sfo1410143707420/sfo1410067598309.html#sfo1410067639578>

Intel (2019). *Introduction to the Hard Processor system* [online]. Santa Clara, California: Intel Corporation [viitattu 07.01.2019]. Saatavissa: <https://www.intel.com/content/www/us/en/programmable/documentation/sfo1410143707420/sfo1410067598309.html#sfo1410067639578>

Kaner, Cem. (1999). *Testing Computer Software*. New York: John Wiley & Sons, Inc. ISBN 0-471-35846-0

Kulov, Ste (2014). *Intro to Programmable Logic & FPGAs* [online]. Pumping Station: One [viitattu 1.10.2018]. Saatavissa: <https://pumpingstationone.org/2014/04/intro-to-programmable-logic-fpgas-april-27/>

LXDE (2018). *LXDE – Desktop Environment for All* [online]. LXDE Contributors [viitattu 8.1.2019]. Saatavissa: < https://lxde.org/>.

Mantere, Timo. (2003). *Automatic Software Testing by Genetic Algorithms.* Väitöskirja. Vaasan yliopisto.

Maxfield, Clive (2012). *Altera's shipping its first SoC FPGAs* [online]. San Francisco, California: Electronic Engineering Times [viitattu 7.11.2018]. Saatavissa: <https://www.eetimes.com/document.asp?doc_id=1317554>

Mazumder, Pinaki & Elizabeth Rudnick. (1998). Genetic Algorithms for VLSI Design, Layout and Test Automation. Upper Saddle River, New Jersey: Prentice Hall. ISBN 978-0130115669.

Pardalos, Panos. (2008). *Pareto Optimality, Game Theory and Equilibria.* Gainesville, Florida: Springer Science + Business Media, LLC. ISBN 978-0-387-77246-2.

Patton, Ron. (2006). *Software Testing*. Indianapolis, Indiana: Sams Publishing. ISBN 0-672-32798-8

Roper, Mark, Iain Maclean, Andrew Brooks, James Miller & Murray Wood (1995). *Genetic Algorithms and the Automatic Generation of Test Data.* Research Report RR/95/195 [EFoCS-19-95]. University of Strathclyde.

Srivastava, Praveen. (2009). Application of Genetic Algorithm in Software Testing. *International Journal of Software Engineering and Its Applications.* 3:4. 87–96.

Sthamer, H.-H. (2003). *The Automatic Generation of Software Test Data Using Genetic Algorithms.* Väitöskirja. University of Glamorgan.

Terasic (2018). *SoCKit - the Development Kit for New SoC Device* [online]. Taiwan: Terasic Inc [viitattu 1.10.2018]. Saatavissa: <https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=816&PartNo=1>

Wilson, Peter (2007). *Design Recipes for FPGAs*. Oxford: MPG Books Ltd. ISBN 978-0-7506-6845-3.

Zwolinski, Mark. (2004). *Digital System Design with VHDL. Second Edition*. Harlow, England: Pearson Education Limited. ISBN 0-130-39985-X

Liitteet

Liite 1. C-lähdekoodit

Tässä esitellään `Genetic.c` ja `Random.c` -koodit, jotka tehtiin tätä diplomityötä varten ja joita käytettiin testauksessa.

Genetic.c

```
//Genetic.c
/*This program file has the whole genetic algorithm
which is used to generate test cases to VHDL programs.*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "Genetic.h"
#include <time.h>
#include "SoCLib.h"

/*INITIALIZE METHODS*/
int crossOver(int p1, int p2); //do crossover
int population(); //initiate population
int mutate(); //mutate
int fitnessFunction(); //calculate fitness function
int sort(); //sort population
int tournament(); //tournament selection
int replace(); //replace worst chromosomes
int print(); //print population
int store(); //store the solution
int print_solution(); //print all the solutions
int fitnessFunction2(); //second fitness function
int sort2(); //sort by second fitness function
int test(); //test method

/*SOLUTION VARIABLES*/
int solution_flag = 0; //solution flag
int solution_loop = 0; //solution loop count
int solution[10][8]; //solution table

/*FPGA VARIABLES*/
int input [8]; //input to FPGA
int output[3] = {0}; //output from FPGA

/*BRANCH VARIABLES*/
int nob = 8; //number of branches
int branch [11]; //branch to compare the new branch
int branch2 [11]; //new branch received from FPGA
int best_branch = 0; //best branch of every iteration
```

```
/*GA VARIABLES*/
static const int popsize=100; //population size
static const int chrom_length=8; //chromosome length
int elitism=2; //number of elitism
int parents=5; //number of pairs of parents
int replace_amount = 6; //amount of worst chromosomes to
replace
int iteration = 10; //amount of iteration times to find 1
solution
int fit_pop; //help variable for Fitness function 2
int pop[100][8]; //population table
int fitness[100]; //fitness table
int p1,p2; //parents

/*RANDOM AND HELP VARIABLES*/
int min=0; //random minimum
int max=100; //random maximum
int mod, dec, fit, check, loop_count = 0; //help variables

int main()
{
    //initialize FPGA conduit and random
    SoC_init();
    srand(time(NULL));

    //do until enough solutions are found
    while(solution_flag==0)
    {
        //initialize population and
        //calculate fitness functions
        population();
        fitnessFunction();
        sort();
        fitnessFunction2();
        sort2();
        print();

        //do amount of times finding one solution
        while(loop_count!=iteration)
        {
            for(int j=0;j<parents;j++)
            {
                p1 = tournament(); //first parent
                p2 = tournament(); //second parent
                while(p1 == p2) //if parents are the same,
change parent2
                {
                    p2 = tournament();
                }
                printf("Parents: %d %d", p1,p2);
```

```
                crossOver(p1,p2); //crossover selected par-
ents
            }

            //do mutation, fitness and replacing
            mutate();
            fitnessFunction();
            sort();
            fitnessFunction2();
            sort2();
            replace();
            fitnessFunction();
            sort();
            fitnessFunction2();
            sort2();
            loop_count++;
        }

        //solution_flag++;

        print();
        printf("loop number: %d\n",loop_count);
        loop_count=0; //reset loop count

        //print out the best chromosome
        printf("Best chromosome: ");
        for(int i=0;i<chrom_length;i++)
        {
            printf("%d ",pop[0][i]);
        }
        printf("\n");

        //just to print fitness
        for(int i=0;i<popsize;i++)
        {
            printf("Fitness: %d\n",fitness[i]);
        }

        //store the best solution
        store();
        //increase the loop
        solution_loop++;

        //print the current branch usage
        printf("\n Branch usage: \n");
        for(int i=0; i<nob; i++)
        {
            printf("%d ",branch[i]);
        }
        printf("\n");

        //check if all the branches are done
        for(int i=0; i<nob; i++)
```

```
        {
            if(branch[i]==1)
            {
                check++;
            }

            //if all the branches are done
            if(check==nob)
            {
                solution_flag=1;
            }
        }
        check=0;
    };
    //print the solutions
    print_solution();

    //close the conduit for FPGA
    printf("Closing the conduit\n");
    SoC_stop();
    printf("The end\n");

    return 0;
}

//method to replace worse chromosomes
int replace()
{
    int r; //random
    for (int i=popsize-replace_amount;i<popsize;i++)
    {
        for(int j=0;j<chrom_length;j++)
        {
            r = (rand() % (max + 1 - min)) + min;
            pop[i][j]=r;
        }
    }
}

//initialize population
int population()
{
    printf("Population \n");
    int r; //random
    for (int i=0;i<popsize;i++)
    {
        for(int j=0;j<chrom_length;j++)
        {
            r = (rand() % (max + 1 - min)) + min;
            pop[i][j]=r;
        }
    }
}
```

```
//method to do crossover
int crossOver(int parent1, int parent2)
{
    int pop_store[chrom_length]; //temporal store array
    int pop_store2[chrom_length]; //temporal store array
    int random; //random variable
    int do_while=0; //variable to do while loop

    do
    {
        random = (rand() % (chrom_length + 1 - min)) + min;
//select crossover point
        for(int i=0;i<random;i++) //make first parts of 2
offsprings
        {
            pop_store[i]=pop[parent1][i]; //first offspring
            pop_store2[i]=pop[parent2][i];   //second   off-
spring
        }

        for(int  i=random;i<chrom_length;i++)  //make  last
parts of 2 offsprings
        {
            pop_store[i]=pop[parent2][i]; //first offspring
            pop_store2[i]=pop[parent1][i];   //second   off-
spring
        }

        do_while=1;
    }while(do_while==0);

    for(int i=0;i<chrom_length;i++) //replace parents with
offspring
    {
        pop[parent1][i]=pop_store[i];
        pop[parent2][i]=pop_store2[i];

    }

}

//method to mutate chromosomes
int mutate()
{
    int random; //random number

    for(int i=elitism;i<popsize;i++) //loop to check popula-
tion
    {
        //generate random number
        random = (rand() % (chrom_length + 1 - min)) + min;;
```

```
            for(int j=0;j<chrom_length;j++)
            {
                //when getting to that random number
                if(j==random)
                {
                    //mutate the part
                    if(pop[i][j]==0)
                    {
                        pop[i][j]=1;
                    }
                    else if(pop[i][j]==1)
                    {
                        pop[i][j]=0;
                    }
                }
            }
        }
    }

    //method to calculate fitness values
    int fitnessFunction()
    {
        //reset fitness values
        for(int i=0;i<popsize;i++)
        {
            fitness[i]=0;
        }

        for(int i=0;i<popsize;i++)
        {

            for(int j=0; j<chrom_length; j++)
            {
                input[j]=pop[i][j];
                SoC_write(input[j], j);
            }

            dec = SoC_read(0); //read variable

            //convert variable to binary
            for (int i=0; i<nob; i++)
            {
                mod = dec % 2;
                branch2[(nob-1)-i]=mod;
                dec = dec / 2;
            }

            //compare new branch to old branch
            for(int i=0; i<nob; i++)
            {
                if(branch[i]==0 && branch2[i]==1)
                {
                    fit++;
```

```
            }
        }

        //insert fitness value to table
        fitness[i]=fit;
        //reset help variables
        fit = 0;
        mod = 0;
        dec = 0;
    }
}

int fitnessFunction2()
{
    //help variable
    fit_pop=0;

    //only calculate the second fitness to
    //chromosomes that have the same best fitnesses
    for(int i=1; i<popsize; i++)
    {
        if(fitness[i]<fitness[0])
        {
            fit_pop=i;
            if(i==(popsize-1))
            {
                fit_pop=i;
            }
            break;
        }
    }

    //calculate the fitness
    for(int i=0; i<fit_pop; i++)
    {
        for(int j=0; j<chrom_length; j++)
        {
            input[j]=pop[i][j];
            SoC_write(input[j], j);
        }
        dec = SoC_read(0); //read variable

        //convert variable to binary
        for (int i=0; i<nob; i++)
        {
            mod = dec % 2;
            branch2[(nob-1)-i]=mod;
            dec = dec / 2;
        }

        //increase fitness
        for(int i=0; i<nob; i++)
        {
```

```
            if(branch2[i]==1)
            {
                fit++;
            }
        }

        fitness[i]=fit; //insert fitness value
        //reset variables
        fit = 0;
        mod = 0;
        dec = 0;
    }
}

//method to store the selected chromosome
int store()
{
    //help print
    printf("\nSTORE");
    //print the current loop
    printf("\nSOLUTION LOOP: %d",solution_loop+1);
    //send and read the branches for selected chromosome
    for(int j=0; j<chrom_length; j++)
    {
        input[j]=pop[0][j];
        SoC_write(input[j], j);
    }

    best_branch = SoC_read(0);

    //calculate the branch
    for (int i=0; i<nob; i++)
    {
        mod = best_branch % 2;
        branch2[(nob-1)-i]=mod;
        best_branch = best_branch / 2;
    }

    //print the old branch
    printf("\nOLD BRANCH: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch[i]);
    }

    //print the new branch
    printf("\nNEW BRANCH: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch2[i]);
    }

    //store the new branch to used branches
```

```
    for(int i=0; i<nob; i++)
    {
        if(branch[i]==0 && branch2[i]==1)
        {
            branch[i]=branch2[i];
        }
    }

    //print current used branches
    printf("\nBRANCH AFTER NEW BRANCH!!: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch[i]);
    }

    //store the used solution to solution table
    printf("\nSolution loop: \n");
    for(int i=0; i<chrom_length; i++)
    {
        solution[solution_loop][i]=pop[0][i];
    }
}

//method to sort population by fitness function 1
int sort()
{
    int swap2; //help variable

    for(int i=0; i<popsize; i++)
    {
        for(int j=0; j<popsize; j++)
        {
            //basic swap sort method
            if(fitness[i] > fitness[j])
            {
                int swap = fitness[i];
                fitness[i] = fitness[j];
                fitness[j] = swap;

                for(int k=0;k<chrom_length;k++)
                {
                    swap2 = pop[i][k];
                    pop[i][k] = pop[j][k];
                    pop[j][k] = swap2;

                }
            }
        }
    }
}

//method to sort population by fitness function 2
int sort2()
```

```
{
    int swap2; //help variable

    for(int i=0; i<fit_pop; i++)
    {
        for(int j=0; j<fit_pop; j++)
        {
            //basic swap sort method
            if(fitness[i] < fitness[j])
            {
                int swap = fitness[i];
                fitness[i] = fitness[j];
                fitness[j] = swap;

                for(int k=0;k<chrom_length;k++)
                {
                    swap2 = pop[i][k];
                    pop[i][k] = pop[j][k];
                    pop[j][k] = swap2;
                }
            }
        }
    }
}

int tournament()
{
    int pop_select = popsize/4; //select quarter of popula-
tion to possible parents
    int select_array[pop_select]; //selection array

    //reset selection array
    for(int i=0; i<pop_select; i++)
    {
        select_array[i]=0;
    }

    int select=0; //help variable
    int random; //random variable
    bool flag = true; //help flag

    //make random number and put it to selection array 0
    random = (rand() % ((popsize-1) - elitism + 1)) + elitism;
    select_array[0]=random;

    for(int i=1;i<pop_select;i++)
    {
        random = (rand() % ((popsize-1) - elitism + 1)) +
elitism; //random
        select_array[i]=random; //put random to select array

        for(int j=0;j<i;j++) //all selected parents
        {
```

```
            if(random==select_array[j]) //if that parent is
already selected
            {
                i--; //select again
                break;
            }
        }
    }

    //do tournament selection
    while(flag)
    {
        flag=false;
        for(int j=0; j<pop_select-1; j++)
        {

            if(fitness[select_array[j]] < fitness[select_ar-
ray[j+1]]) //tournament selection
            {
                int swap = select_array[j];
                select_array[j] = select_array[j+1];
                select_array[j+1] = swap;
                flag=true;
            }
        }
    }

    select=select_array[0]; //best parent is selected
    return select;
}

//method to print population
int print()
{
    for(int i=0;i<popsize;i++)
    {
        for(int j=0;j<chrom_length;j++)
        {
            printf("%d ",pop[i][j]);
        }
        printf("\n");
    }
}

//method to print solutions
int print_solution()
{
    printf("\nSolutions: \n");
    //do as many time as there are solutions
    for(int i=0;i<solution_loop;i++)
    {
        //print the solutions
        printf("\nSolution number: %d\n",i+1);
```

```
            for(int k=0; k<chrom_length; k++)
            {
                printf("%d ",solution[i][k]);
            }

            //check the solutions from FPGA
            printf("\nUsed branches: \n");
            for(int j=0; j<chrom_length; j++)
            {
                input[j]=solution[i][j];
                SoC_write(input[j], j);
            }

            best_branch = SoC_read(0);
            printf("\n");
            printf("Branch decimal: %d\n", best_branch);

            //calculate the used branches
            for (int l=0; l<nob; l++)
            {
                mod = best_branch % 2;
                branch2[(nob-1)-l]=mod;
                best_branch = best_branch / 2;
            }

            //print the branches
            for(int m=0; m<nob; m++)
            {
                printf("%d ",branch2[m]);
            }
        }
    }
```

Random.c

```
/*Random.c
This program file has a random generator
which is used to generate test cases for a VHDL program.*/

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "Genetic.h"
#include <time.h>
#include "SoCLib.h"

/*INITIALIZE METHODS*/
int do_random(); //initiate population
int store(); //store the solution
int print_solution(); //print all the solutions

/*SOLUTION VARIABLES*/
int solution_flag = 0; //solution flag
int solution_loop = 0; //solution loop count
int solution[10][8]; //solution table

/*FPGA VARIABLES*/
int input [8]; //input to FPGA
int output[3] = {0}; //output from FPGA

/*BRANCH VARIABLES*/
int nob = 8; //number of branches
int branch [11]; //branch to compare the new branch
int branch2 [11]; //new branch received from FPGA
int best_branch = 0; //best branch of every iteration

/*RANDOM VARIABLES VARIABLES*/
static const int popsize=100; //population size
static const int chrom_length=8; //chromosome length
int pop[100][8]; //population table

/*RANDOM AND HELP VARIABLES*/
int min=0; //random minimum
int max=100; //random maximum
int mod, dec, fit, check, loop_count = 0; //help variables

int main()
{
    //initialize FPGA conduit and random
    SoC_init();
    srand(time(NULL));

    //do until enough solutions are found
    while(solution_flag==0)
    {
```

```
        do_random();
        printf("Best chromosome: ");
        for(int i=0;i<chrom_length;i++)
        {
            printf("%d ",pop[0][i]);
        }
        printf("\n");

        //store the best solution
        store();
        //increase the loop
        solution_loop++;

        //print the current branch usage
        printf("\n Branch usage: \n");
        for(int i=0; i<nob; i++)
        {
            printf("%d ",branch[i]);
        }
        printf("\n");

        //check if all the branches are done
        for(int i=0; i<nob; i++)
        {
            if(branch[i]==1)
            {
                check++;
            }

            //if all the branches are done
            if(check==nob)
            {
                solution_flag=1;
            }
        }
        check=0;
    };
    //print the solutions
    print_solution();

    //close the conduit for FPGA
    printf("Closing the conduit\n");
    SoC_stop();
    printf("The end\n");

    return 0;
}

//initialize population
int do_random()
{
    printf("Random \n");
    int r; //random
```

```
    for(int j=0;j<chrom_length;j++)
    {
        r = (rand() % (max + 1 - min)) + min;
        pop[0][j]=r;
    }
}

//method to store the selected chromosome
int store()
{
    //help print
    printf("\nSTORE");
    //print the current loop
    printf("\nSOLUTION LOOP: %d",solution_loop+1);
    //send and read the branches for selected chromosome
    for(int j=0; j<chrom_length; j++)
    {
        input[j]=pop[0][j];
        SoC_write(input[j], j);
    }

    best_branch = SoC_read(0);
    //calculate the branch
    for (int i=0; i<nob; i++)
    {
        mod = best_branch % 2;
        branch2[(nob-1)-i]=mod;
        best_branch = best_branch / 2;
    }

    //print the old branch
    printf("\nOLD BRANCH: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch[i]);
    }
    //print the new branch
    printf("\nNEW BRANCH: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch2[i]);
    }

    //store the new branch to used branches
    for(int i=0; i<nob; i++)
    {
        if(branch[i]==0 && branch2[i]==1)
        {
            branch[i]=branch2[i];
        }
    }

    //print current used branches
```

```
    printf("\nBRANCH AFTER NEW BRANCH!!: \n");
    for(int i=0; i<nob; i++)
    {
        printf("%d ", branch[i]);
    }

    //store the used solution to solution table
    printf("\nSolution loop: \n");
    for(int i=0; i<chrom_length; i++)
    {
        solution[solution_loop][i]=pop[0][i];
    }
}
//method to print solutions
int print_solution()
{
    printf("\nSolutions: \n");
    //do as many time as there are solutions
    for(int i=0;i<solution_loop;i++)
    {
        //print the solutions
        printf("\nSolution number: %d\n",i+1);
        for(int k=0; k<chrom_length; k++)
        {
            printf("%d ",solution[i][k]);
        }

        //check the solutions from FPGA
        printf("\nUsed branches: \n");
        for(int j=0; j<chrom_length; j++)
        {
            input[j]=solution[i][j];
            SoC_write(input[j], j);
        }

        best_branch = SoC_read(0);
        printf("\n");
        printf("Branch decimal: %d\n", best_branch);

        //calculate the used branches
        for (int l=0; l<nob; l++)
        {
            mod = best_branch % 2;
            branch2[(nob-1)-l]=mod;
            best_branch = best_branch / 2;
        }
        //print the branches
        for(int m=0; m<nob; m++)
        {
            printf("%d ",branch2[m]);
        }
    }
}
```

Liite 2. VHDL-lähdekoodit

Tässä esitellään testattavat ohjelmat `First_case.vhd`, `Second_case.vhd`, `Third_case.vhd`, `Fourth_case.vhd` ja `Fifth_case.vhd` seurattavien haarojen lisäämisen jälkeen.

First_case.vhd

```
first_case : process(rec_data_logic)
variable fitness : integer range 0 to 100000;

begin

--reset fitness and branch table
fitness := 0;
init_loop : for i in 0 to 5 loop
     br_1(i) <= 0;
end loop;

-- insert logic values
INA1 <= rec_data_logic(0);
INA2 <= rec_data_logic(1);
INA3 <= rec_data_logic(2);
INA4 <= rec_data_logic(3);

--insert branches to program
O1 <= INA1 and INA2;
if O1 = '1' then
          br_1(0) <= 1;
else
     br_1(1) <= 1;
end if;

O2 <= INA3 and INA4;
if O2 = '1' then
     br_1(2) <= 1;
else
     br_1(3) <= 1;
end if;

 O3 <= O1 or O2;
if O3 = '1' then
     br_1(4) <= 1;
else
     br_1(5) <= 1;
end if;

-- calculate bit to decimal
```

```
calculate : for i in 0 to 5 loop
fitness := fitness + (br_1(i) * (2**(5-i)));
end loop;

--send decimal value to genetic algorithm
--AMMS_data_to_ARM(0) <= std_logic_vector(to_signed(fitness,
32));

end process;
```

Second_case.vhd

```
second_case : process(rec_data_logic2)
--variable fitness : integer range 0 to 100000;
variable l : integer range 0 to 1;

begin

--while loop variable
l := 0;
--reset fitness and branch table
fitness := 0;
init_loop : for i in 0 to 5 loop
     br_2(i) <= 0;
end loop;

-- insert logic values
INAA1 <= rec_data_logic2(0);
INAA2 <= rec_data_logic2(1);
INAA3 <= rec_data_logic2(2);
INAA4 <= rec_data_logic2(3);
INOO1 <= rec_data_logic2(4);
INOO2 <= rec_data_logic2(5);

--insert branches to program
while ( l = 0) loop
     OO1 <= INAA1 and INAA2;
     if OO1 = '1' then
               br_2(0) <= 1;
               exit;
     else
          br_2(1) <= 1;

     end if;

     OO2 <= INAA3 and INAA4;
     if OO2 = '1' then
          br_2(2) <= 1;
          exit;
     else
          br_2(3) <= 1;
     end if;

     OO3 <= INOO1 or INOO2;
     if OO3 = '1' then
          br_2(4) <= 1;
          exit;
     else
          br_2(5) <= 1;
     end if;

     l := 1;
```

```
end loop;

-- calculate bit to decimal
calculate : for i in 0 to 5 loop
fitness := fitness + (br_2(i) * (2**(5-i)));
end loop;

--send decimal value to genetic algorithm
AMMS_data_to_ARM(0)  <=  std_logic_vector(to_signed(fitness,
32));

end process;
```

Third_case.vhd

```
third_case : process(rec_data2)
variable bubble : bubble_array := (0,0,0,0,0,0,0,0);
variable swap : integer range 0 to 100;
variable fitness : integer range 0 to 100000;
variable flag : integer range 0 to 10;
variable flag2 : integer range 0 to 10;

begin
--reset branches
init_loop : for i in 0 to 7 loop
     br_3(i) <= 0;
end loop;

--insert values to variable array
init_loop2 : for i in 0 to 7 loop
     bubble(i) := rec_data1(i);
end loop;

br_3(0) <= 1;

fitness := 0;
swap := 0;
flag := 0;
flag2:= 0;


first_loop : for i in 0 to 7 loop
     br_3(1) <= 1;
     flag := flag+1;
     second_loop : for j in 0 to 7-i-1 loop
          br_3(2) <= 1;
          flag2 := flag2+1;
          --if next number is greater
          if bubble(j) > bubble(j+1) then
               br_3(3) <= 1;
               --swap
               swap := bubble(j);
               bubble(j) := bubble(j+1);
               bubble(j+1) := swap;
          else
               br_3(4) <= 1;
          end if;
          --if looped back
          if flag2 > 1 then
               br_3(5) <= 1;
          end if;
     end loop;
     flag2 := 0;
     --if looped back
     if flag > 1 then
```

```
          br_3(6) <= 1;
     end if;
end loop;
flag := 0;
br_3(7) <= 1;

--calculate fitness
calculate : for i in 0 to 7 loop
     fitness := fitness + (br_3(i) * (2**(7-i)));
end loop;

AMMS_data_to_ARM(0)  <=  std_logic_vector(to_signed(fitness,
32));


end process;
```

Fourth_case.vhd

```
fourth_case : process(rec_data2)
--variables
variable fitness : integer range 0 to 100000;
variable flag2 : integer range 0 to 10;
variable flag3 : integer range 0 to 10;

begin
init_loop : for i in 0 to 10 loop
     br_4(i) <= 0;
end loop;

br_4(0) <= 1;
--for loop variables
flag <= 0;
flag2 := 0;
flag3 := 0;
fitness := 0;

first_loop : for i in 0 to 7 loop
     br_4(1) <= 1;
     --if true
     if rec_data2(i) = 1 then
          br_4(2) <= 1;
          second_loop : for j in i+1 to 7 loop
               br_4(3) <= 1;
               flag3 := flag3+1;
               --if not allowed
               if rec_data2(j) = 1 and rule_data(i, j) = 0
then
                    br_4(4) <= 1;
                    flag <= 1;
                    exit first_loop;
                    exit second_loop;
               else
                    br_4(8) <= 1;
               end if;
               --check if looped back
               if flag3 > 1 then
                    br_4(9) <= 1;
               end if;
          end loop;
          br_4(10) <= 1;
          flag3 := 0;
     else
          br_4(5) <= 1;
     end if;
     --check if looped back
     if flag2 > 1 then
          br_4(6) <= 1;
     end if;
```

```
end loop;
flag2 := 0;
if flag = 0 then
     br_4(7) <= 1;
end if;

--calculate fitness
calculate : for i in 0 to 10 loop
     fitness := fitness + (br_4(i) * (2**(10-i)));
end loop;

AMMS_data_to_ARM(0)  <=  std_logic_vector(to_signed(fitness,
32));


end process;
```

Fifth_case.vhd

```
calculate_class : process(featurebuffer)

variable ready:    natural := 0;
variable fitness : integer range 0 to 100000;

begin

init_loop : for i in 0 to 8 loop
     br_5(i) <= 0;
end loop;

     fitness := 0;
     ready := 0;
     br_5(0) <= 1;

     if featurebuffer(4) < 2461 then
          br_5(1) <= 1;
          class <= 1;
          ready := 1;

     else
          br_5(2) <= 1;

     end if;

     if ((featurebuffer(10)) < 4116) and (ready = 0) then

          br_5(3) <= 1;
          class <= 2;
          ready := 1;


     elsif (ready = 0) then
          br_5(4) <= 1;

     end if;

     if((featurebuffer(11)) > 5148) and (ready = 0) then

          br_5(5) <= 1;
          class <= 5;
          ready := 1;


     elsif (ready = 0) then
          br_5(6) <= 1;

     end if;
```

```
     if ((featurebuffer(6)) < 8474) and (ready = 0) then

          br_5(7) <= 1;
          class <= 3;
          ready := 1;

     elsif (ready = 0) then
          br_5(8) <= 1;
          class <= 4;
          ready := 1;

     end if;

     ready := 0;


     --calculate fitness
     calculate : for i in 0 to 8 loop
          fitness := fitness + (br_5(i) * (2**(8-i)));
     end loop;

AMMS_data_to_ARM(0)<=std_logic_vector(to_signed(fitness,
32));



end process;
```