UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

AUTOMATION TECHNOLOGY

Staffan Järn

IMPLEMENTATION OF GENETIC ALGORITHMS ON AN FPGA-ETHERNET TESTER

Master's thesis for the degree of Master of Science in Technology submitted for evaluation.

Vaasa 13.5.2014

Supervisor	Jarmo Alander
Instructor	Petri Välisuo

FOREWORD

This Master's thesis was initiated by the TehoFPGA project group at the University of Vaasa. The thesis concerns FPGA hardware implementations using Genetic Algorithms.

I would like to thank my supervisor Professor Jarmo Alander at the University of Vaasa, for providing me to participate in this project. Also special thanks to my colleague Olli Rauhala for great guidance and cooperation in this project. Also, not to forget Mika Ruohonen that has provided us with his wide knowledge about the GOOSE protocol and configuration of protection relays.

Finally, very special thanks to my fellow companion Mathias Björk for creative discussions during our coffee and lunch breaks.

Vaasa 13.5.2014

Staffan Järn

TABLE OF CONTENTS **page**

SYMBOLS AND ABBREVIATIONS

CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detect
DSAP	Destination Service Access Point
DUT	Device-Under-Test
EC	Evolutionary Computing
FPGA	Field Programmable Gate Array
FPS	Fitness Proportional Selection
GA	Genetic Algorithm
GOOSE	Generic Object Oriented Substation Events
HDL	Hardware Description Language
ICBF	Idle-cycles Between Frames
IEC	International Electrotechnical Commission
IED	Intelligent Electronic Device
LAN	Local Area Network
LFSR	Linear-Feedback Shift Register
LLC	Logical Link Control
LSB	Least Significant Bit
MAC	Media Access Control
OSI	Open Systems Interconnect
PLD	Programmable Logic Device
RTL	Register Transfer Level
SAS	Substation Automation Systems
SFD	Start of Frame Delimiter

SSAP	Source Service Access Point
SUS	Stochastic Universal Sampling
TCP/IP	Transmission Control Protocol / Internet Protocol
TSE	Triple-Speed Ethernet
WAN	Wide Area Network
VHDL	Very High Speed Integrated Circuit HDL (see HDL)
VLAN	Virtual Local Area Network

UNIVERSITY OF VAASA
Faculty of Technology
Author: Staffan Järn
Topic of the Thesis: Implementation of Genetic Algorithms on an FPGA-Ethernet Tester
Supervisor: Professor Jarmo Alander
Instructor: Dr Petri Välisuo
Degree: Master of Science in Technology
Degree Programme: Degree Programme in Electrical and Energy Engineering
Major of Subject: Automation Technology
Year of Entering the University: 2010
Year of Completing the Thesis: 2014 **Pages:** 65

ABSTRACT

In electrical substation automation systems (SAS), intelligent electronic devices (IED) communicate over Ethernet within the IEC 61850 standard. The main objective of the standard is to bring compatibility, security and robustness between different IEDs, regardless the manufacturer. A typical SAS consists of IEDs such as circuit breakers, protection relays and controllers.

This thesis concerns the generation and transmission of Ethernet traffic from a Field Programmable Gate Array (FPGA) to an IED. The research question was to study the robustness of the IEC 61580 standard implementation on an IED and search for Ethernet data that might be harmful for the device. An FPGA, with high speed performance due to its parallelism, combined with a genetic algorithm search optimization process, was chosen to approach the problem. Genetic algorithms are optimization methods which have taken inspiration from biology, where species strive for survival. In this research case, genetic algorithms were implemented in an FPGA where they are adapted to the Ethernet frames by means of recombination and mutation of binary data. Transmission-round trip time feedbacks were measured by an external device, where a larger transmission time results in a greater fitness value, gaining a higher probability of finding harmful data.

The result was a hardware implementation of genetic algorithms on an FPGA platform that manages to transmit Ethernet frames at high speed. In the research it was also obtained that it is possible to cause an IED to crash depending on Ethernet frame data and transmission speed.

KEYWORDS: FPGA, Genetic Algorithms, Ethernet, IEC 61850

VAASAN YLIOPISTO
Teknillinen tiedekunta
Tekijä: Staffan Järn
Diplomityön nimi: Geneettisten algoritmien implementaatio FPGA Ethernet testerillä
Valvojan nimi: Professori Jarmo Alander
Ohjaajan nimi: TkT Petri Välisuo
Tutkinto: Diplomi-insinööri
Koulutusohjelma: Sähkö- ja energiatekniikan koulutusohjelma
Suunta: Automaatiotekniikka
Opintojen aloitusvuosi: 2010
Diplomityön valmistusvuosi: 2014 **Sivumäärä:** 65

TIIVISTELMÄ

Sähköverkon ala-asemien automaatiojärjestelmissä (SAS) älykkäät elektroniikkalaitteet (IED) kommunikoivat IEC 61850 standardin mukaan Ethernetväylällä. Standardin päätavoitteet ovat yhteensopivuus, turvallisuus ja vikasietoisuus eri IED:n välillä, valmistajasta riippumatta. Tyypillinen SAS koostuu IED:stä kuten katkaisijoista, suojareleistä ja ohjaimista.

Tämä diplomityö koskee Ethernet-liikenteen tuottamista ja lähettämistä FPGA:sta IED:hen. Tutkimuksen aiheena oli tutkia IEC 61850 standardin toteutuksen vikasietoisuutta, sekä etsiä IED:lle haitallisia Ethernet-paketteja. Lähestymistapana ongelmaan valittiin FPGA, jolla on rinnakkaisuuden johdosta suuri laskentakapasiteetti, yhdistettynä geneettiseen etsintäalgoritmiin. Geneettiset algoritmit ovat optimointimenetelmiä jotka ovat saaneet vaikutteita biologiasta, missä lajit pyrkivät sopeutumaan selviytyäkseen. Tässä tutkimustapauksessa geneettiset algoritmit muokkaavat binäärisiä Ethernet-kehyksiä rekombinaatiolla ja mutaatiolla. Ulkoisen laitteen avulla lasketaan viestien vasteaika siten että pidempi aika antaa paremman hyvyysarvon ja täten suuremman todennäköisyyden löytää haitallisia kehyksiä.

Saatu tulos oli laitteisto-ohjelmoitu FPGA sovellus geneettisellä algoritmilla, mikä pystyy lähettämään Ethernet-kehyksiä suurella nopeudella. Tutkimuksessa pystyttiin myös kaatamaan IED, riippuen Ethernet-kehysten sisällöstä ja niiden lähettämisnopeudesta.

AVAINSANAT: FPGA, Geneettiset algoritmit, Ethernet, IEC 61850

VASA UNIVERSITET
Tekniska fakulteten
Författare: Staffan Järn
Diplomarbetets titel: Implementering av genetiska algoritmer på en FPGA Ethernet testare
Övervakare: Professor Jarmo Alander
Handledare: TkD Petri Välisuo
Examen: Diplomingenjör
Utbildningsprogram: Utbildningsprogrammet för elektro- och energiteknik
Inriktning: Automationsteknik
Årtal för inledande av studier: 2010
Diplomarbetet färdigställt: 2014 **Sidantal:** 65

ABSTRAKT

I Substation Automation Systems (SAS) kommunicerar intelligenta elektroniska apparater (IED) över Ethernet inom IEC 61850 standarden. Huvudidén med standarden är att tillföra kompabilitet, säkerhet och tillförlitlighet mellan olika IED apparater, oberoende av tillverkare. Ett typiskt SAS består av olika IED enheter som t.ex. kretsbrytare, skyddsrelän och styrenheter.

Denna avhandling behandlar generering och sändning av Ethernet trafik från en Field Programmable Gate Array (FPGA) till en IED. Tanken med avhandlingen var att studera IEC 61850 standardens implementations robusthet på en IED och söka efter Ethernet data som kan vara skadlig för enheten. En FPGA, som har hög prestanda på grund av dess parallellism, kombinerat med sökoptimeringsprocessen genetiska algoritmer valdes som tillvägagångssätt. Genetiska algoritmer är en forskningsgren som tagit inspiration från biologin och arternas strävan för överlevnad. I denna forskning implementerades genetiska algortimer på en FPGA och tillämpades på Ethernet paketen med hjälp av rekombination och mutation av binär data. Med hjälp av återkoppling från en extern enhet beräknades svarstiden för Ethernet paketen, där en längre sändningstid resulterar i ett högre lämplighetsvärde och ökar sannolikheten för att finna skadlig data.

Resultatet var en hårdvaru-implementering av genetiska algoritmer på en FPGA plattform som sänder Ethernet paket i hög hastighet. Inom forskningen observerades också möjligheten att orsaka haveri för funktionen av en intelligent elektronisk apparat beroende på Ethernet paketets innehåll och sändningshastighet.

NYCKELORD: FPGA, Genetiska algoritmer, Ethernet, IEC 61850

1. INTRODUCTION

This Master's thesis was initiated by the Automation technology department at the University of Vaasa, as a part of the research project *TehoFPGA*. The aim of the research project is to enhance the knowledge about FPGA technology in the Vaasa region. The Vaasa region is an internationally important industrial energy cluster, for instance in the making of measurement devices such as protection relays and frequency converters. The TehoFPGA research group is a co-operation between Universities in the region and local companies such as ABB, Wärtsilä, Vamp and Wapice. In the Technobothnia laboratory at Palosaari campus area, a high-tech *DEMVE* laboratory environment consisting of protection devices and monitoring systems from several manufacturers has been developed for enhancing the knowledge about the IEC 61850 standard for students, staff and the industry. (TehoFPGA-I 2012)

In substation automation systems (SAS), communication is executed within the IEC 61850 standard. Regardless of manufacturer, several intelligent electronic devices (IED) in a SAS are compatible to communicate by the Generic Object Oriented Substation Events (GOOSE) protocol. (IEC 61850-1: 9-11)

The aim of this thesis was to study the robustness of devices that communicate with the IEC 61850 standard. The goal was to find packages that can be harmful to a network and in the worst case cause a device to malfunction. By constantly transmitting randomly generated GOOSE messages over the network, the idea was that sooner or later harmful packages might be observed. For this purpose, Genetic Algorithms (GA) seemed to be a potential search method. The choice of using an FPGA as a transmitting device was mainly because of its speed and parallelism. By simultaneously generating several random packages and transmitting them sequentially over the network will improve the speed, compared to a simple microprocessor.

A hardware application was programmed on an Altera DE4 FPGA platform, where the FPGA generates and modifies the payload of Ethernet frames and finally transmits the frames as a flood over the network. Random number generation is implemented with

Linear Feedback Shift Registers (LFSR) and the genetic algorithm operations are executed by logical functions. The system structure consists of the FPGA, a device under test (DUT) and an embedded Raspberry Pi computer, all connected to an Ethernet switch. The Raspberry Pi communicates mutually with the DUT by GOOSE structured messages and measures the latency of the DUT in form of a feedback value, while the FPGA is transmitting data. The feedback value is a mean value of the DUT's latency time depending on how many times the GOOSE message was exchanged. The feedback value, belonging to a specific frame, is sent from the Raspberry Pi to the FPGA via RS-232 communication. Depending on feedback values, the frames are sorted within the FPGA hardware and processed with genetic algorithms for searching the most harmful frames.

The result was a working frame generator that manages to flood frames over the Ethernet, receive feedback from the network, and generate new GA modified frames. However, the presence of the GA did not improve to affect the network in a bad sense. Instead, it was possible to put the DUT into a faulty state depending on other parameters, such as destination MAC address and transmission speed.

This thesis is divided into five chapters, excluding the introduction. Chapter 2 presents *Evolutionary Computing*, its history and basics. The procedures about GA are deeper explained to give the reader basic knowledge about the most common GA operators, such as fitness, selection, recombination and mutation. Chapter 3 presents theory about *Ethernet*, where the Ethernet frame is described in detail, since it's only the payload of the Ethernet frame that will be the essential part in the hardware application. IEC 61850 and GOOSE is only mentioned with basic facts and figures, since it's beyond the scope of this thesis and would demand too many details for this thesis. Chapter 4 presents the *Field Programmable Gate Array* (FPGA). Benefits and performance of the FPGA are discussed and basics about the Hardware Description Language *Verilog* are presented. In chapter 5 the project work is described thoroughly. Structure of the complete system, structure and behaviour of individual modules, behavioural simulations and field tests are presented. The last chapters, conclusions and discussion, will present the achieved results, difficulties during the project and ideas for further development.

2. EVOLUTIONARY COMPUTING

In the 19th century Charles Darwin presented a theory about the biological evolution of species, where natural selection plays a central role. Natural selection favours those species that are adapting to an environment the best way, giving name to the *survival of the fittest* theory. In computer science, evolutionary computing (EC) is a research branch that has taken inspiration from the biology and the process of natural evolution. Computers provide speed and accuracy in handling large amount of data, and also efficiency in repetitive routines. The powerful evolution in nature, where species fight for survival, relate to evolutionary computing by a particular style of problem solving called *trial-and-error*. In an environment, filled with a population of individuals, the priority of the individuals lies in survival and propagation. Depending on the environment, the probability that the individuals survive and multiply is determined by their *fitness*. The higher the fitness, the greater the individual contender. Concerning trial-and-error, given a population of candidates, a greater contender has a larger probability to be selected and used as seed for further candidate solutions.

Diving a little bit deeper into the microscopic view of the natural evolution, the dealing with genetics arises. According to Eiben & Smith: ”The fundamental observation from genetics is that each individual is a dual entity: its phenotypic properties (outside) are represented at a low genotypic level (inside). In other words, an individual’s genotype encodes its phenotype.” The genotype therefore contains information needed to form the particular phenotype, but also the environment has effect on the phenotype. The complete genetic information about the individual is called *genome*.

Mutation of genes or recombination of genes causes genotypic variations which also cause phenotypic variations. The combination of features from two individuals, called *crossover*, results in an offspring that has mixed chromosomes from each parent individuals. (Eiben & Smith 2007: 1-6)

2.1. History of Evolutionary Computing

The history of evolutionary computing dates back to the 1930s with the ideas of Sewell Wright. According to De Jong: “He found it useful to visualize an evolutionary system as exploring a multi-peaked fitness landscape and dynamically forming clusters around peaks of high fitness”. The idea of using an evolutionary system as an optimization process is even today the most common application area of EC. Later on in the 1960s, when computers were further developed, three different fields in EC arose; *evolutionary programming*, *genetic algorithms* and *evolution strategies*. Several years later, in the beginning of 1990s, a fourth branch called *genetic programming*, following the same evolutionary ideas, were added to the field. For long, these fields were developed separately until they finally merged into a common term called *evolutionary algorithms*. (Eiben & Smith 2007: 2; De Jong 2006: 23-24)

2.2. Evolutionary Algorithm

All kinds of different evolutionary algorithms follow the same basic idea. In an environment with a population of individuals, all individuals strive for survival. This is what goes by the name *survival of the fittest* and causing a rise in the fitness of the population. By randomly creating a set of population and evaluating the fitness value of each individual by a *fitness function*, the best solutions are chosen to seed the next generation. Applying recombination within two candidates (parents), result in one or more offspring (children). After a mutation is applied to the offspring it will result in a new candidate that will be competing against old candidates. This process is repeated until a sufficient solution is found or if the computational limit is reached. It is however important to notice that no evolutionary algorithm is optimal in all situations and for all optimization problems, which is called the *no free lunch theorem*. Theoretically there is no super-algorithm that is able to solve all kinds of complex problems. (Eiben & Smith 2007: 15-16; Alander 2006: 17)

```
BEGIN
    INITIALIZE population randomly;
    EVALUATE each candidate;
    REPEAT UNIL (TERMINATION CONDITION is satisfied) DO
        1 SELECT parents;
        2 RECOMBINE pairs of parents;
        3 MUTATE the resulting offspring;
        4 EVALUATE new candidates;
        5 SELECT individuals for the next generation;
    OD
END
```

Figure 1. Pseudocode of an evolutionary algorithm. (Adapted from Eiben & Smith 2007: 16)

2.3. Genetic Algorithms

Genetic algorithms are one of the most common types of evolutionary algorithms for function optimization problems, automatic programming, machine learning, economics etc. Since there is no single way of defining a genetic algorithm, the most important decision is how to best represent a candidate solution to the particular application. The simplest representation is with binary values but also integer, floating-point and permutation representations can be more suitable in some applications. (Mitchell 1998: 15-16)

In the classical genetic algorithm, also called the "simple GA", individuals have binary representation and fitness evaluation, low probability of mutation, and recombination of genes that guides the generation of new candidate solutions. A random initial population is evaluated with fitness values for each individual. According to the fitness value, a probability is given that this individual is chosen as a parent. The number of parents is the same as the population size and therefore the same individual can be copied as a parent many times. The selected individuals are paired at random and mutually swapped at a crossover point, building new offspring individuals. Finally, every bit position in the offspring is generated a random number in the range [0,1] and compared to a very low mutation rate value (e.g. 0.05). If the random number is below the mutation rate, the binary value in that bit position is flipped. (Eiben & Smith 2007: 37-38)

2.3.1. Recombination

Recombination is one of the most important diversity features in genetic algorithms. A randomly initialized population, recombination of parents, and mutation of offspring creates diversity within the population, where the diversity is important in finding new solutions. *Crossover* is often used as another term for recombination between parents. Below follows a short presentation of different basic crossover operations applied from Eiben & Smith (2007: 47-49).

One-point crossover

A random number in the range [*0, l-1*] is selected (where *l* is the length of the genome) and splitting the parents at that point. New children are generated with switched tails.

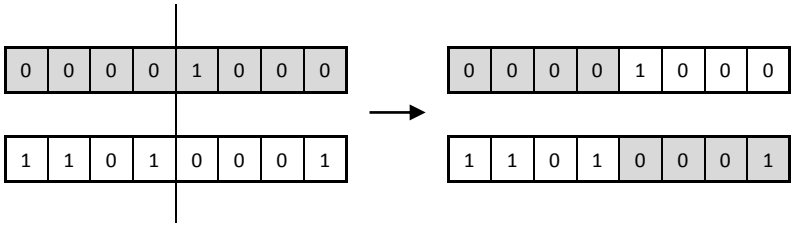


Figure 2. One-point crossover. (*l = 4*)

N-point crossover

Similar to the one-point crossover, but with *n* random crossover points.

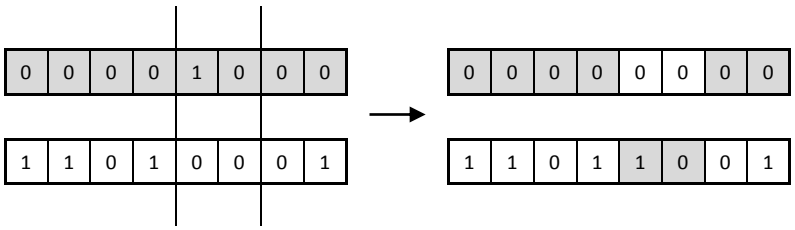


Figure 3. N-point crossover. (*l =4, n =2*)

Uniform crossover

A randomly generated crossover index vector is determining the genes that will be swapped between the parents. The crossover index vector is a binary string containing zeros and ones. If the binary value equals one, swapping between parent genes occurs.

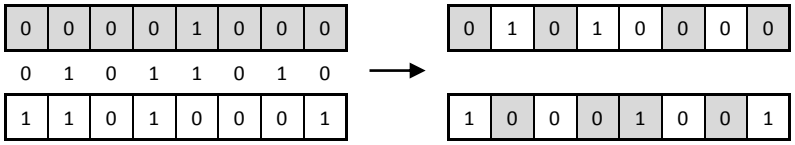


Figure 4. Uniform crossover. (*crossover index vector: [0 1 0 1 1 0 1 0]*)

One-point- and n-point crossover has however proven to introduce too weak variance to the offspring when the parent population is fairly heterogeneous. If a fixed number of crossover points are always chosen, the probability grows that nearby genes are combined as a group rather than widely separated, causing a distance bias. The distance bias can be reduced by adding more crossover points, but along with that follows increasing disruptiveness. By using uniform crossover the variation is increased and the distance bias can be neglected by controlling the probability rate. This is one of the main reasons that uniform crossover is one of the most widely used recombination methods. (De Jong 2006: 64-65)

2.3.2. Mutation

Mutation is an operation within an offspring, causing minor genetic changes to the offspring. The mutation probability is determined by the mutation rate, often small not to cause to large changes to the individual properties. Below follows a short presentation of different basic mutation operations applied from Eiben & Smith (2007: 43-46).

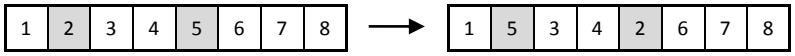
Bitwise mutation

The most common binary mutation operator. With a small mutation probability a bit is flipped from 0 to 1, or 1 to 0.

Figure 5. Bitwise mutation.

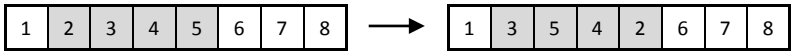
Considering integer representations there are a few operations presented in the figure 6 below.

Swap mutation



Insert mutation

Scramble mutation



Inversion mutation

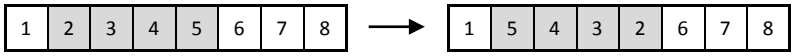


Figure 6. Mutation operators for integer representation. *Swap mutation*: bits at specified positions are mutually swapped. *Insert mutation*: bits at specified positions inserted into new point in offspring. *Scramble mutation*: a set of bits randomly arranged into new point in offspring. *Inversion mutation*: a set of bits inverted into new point in offspring.

2.3.3. Selection

There are two main types of selection mechanisms in GA: stochastic and deterministic selection methods. In the stochastic selection individuals in the selection pool are a given a probability *p* of being chosen. The drawback of using stochastic selection is that the best individual in the population may never get chosen and the worst individual selected multiple times, while using a deterministic selection algorithm each individual is forced to be chosen exactly once. The best way of choosing selection mechanism for a specific application is to visualize how the mechanism is distributing the selection probability over the selection pool. (De Jong 2006: 54-55). In this chapter some of the most common fitness selection methods are presented below.

Fitness Proportional Selection

In Fitness Proportional Selection (FPS) the selection probability depends on the fitness value of the individual compared to the total fitness value of the population. A simple example of the FPS is illustrated in table 1 below.

String	Population	Fitness **x**	f(x) = $\mathbf{x}^2$	$Prob_i$	Exp. count	Act. count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4	4

Table 1. Fitness Proportional Selection. Maximizing values of x^2 for *x* in the range 0-31 (population size = 5 bits). Individuals with high fitness obtain a higher probability of getting chosen. In this stochastic case the most fit is chosen twice, while the least fit is not chosen at all. Reference from Eiben & Smith (2007: 39).

However, some main drawbacks with this selection method have been recognized. Outstanding individuals that are much better than the rest can take over the entire population quickly and fitness values close together can cause almost uniformly random selec-

tion. These problems can still be avoided with the help of windowing or with sigma scaling. (Mitchell 1998:167-168)

Fitness Proportional Selection with Roulette Wheel

In FPS with Roulette Wheel selection each individual is assigned a slice of a circular roulette wheel, where the size of the slice is proportional to the fitness of the individual. The roulette wheel is spun *n* times, where *n* is the number of individuals in the population. At every spin the wheel stops and the marker lands at an individual, that individual is chosen for next generation. With really bad luck, even the weakest individual can be chosen randomly to represent all of the new individuals. That scenario can be prevented by the use of stochastic universal sampling (SUS) to minimize the spread. Instead of spinning the wheel *n* times, the wheel is spun only once, but with *n* equally spaced pointers that will list all selection probabilities. (Mitchell 1998: 166-167, Eiben & Smith 2007: 62)

Ranking selection

The drawbacks in the previous fitness proportional selection can also be avoided with ranking selection. The population is sorted on basis of fitness and given a rank value. Selection is then done depending on rank, rather than directly from the fitness. The mapping from rank to selection probability can be either linear or exponential. (Mitchell 1998: 169-170, Eiben & Smith 2007: 60-61)

With linear ranking the selection pressure is limited, regardless the value of *s*:

$$P(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}, \quad (1)$$

where μ is the rank and *s* ($1.0 < s \leq 2.0$).

If more emphasis is needed on selecting individuals of above-average fitness, a higher selection pressure on those is obtained with the exponential rank:

$$P(i) = \frac{(1-e^{-i})}{c}, \quad (2)$$

where *c* is a normalisation factor so that the sum of probabilities is unity.

Table 2. Ranking selection methods. Where P_{selFP} = fitness proportional selection, P_{selLR} = linear ranking selection and P_{selEXP} = exponential ranking selection.

Ind	Fitness	Rank	P_{selFP}	P_{selLR} (s=2)	P_{selEXP}
A	5	3	0.19	0.13	0.18
B	4	4	0.15	0.2	0.18
C	9	6	0.33	0.33	0.18
D	6	5	0.22	0.27	0.18
E	2	2	0.07	0.07	0.16
F	1	1	0.04	0	0.12
Sum	27	$\bar{x}$ = 3.5	1.0	1.0	1.0

Tournament selection

Tournament selection suits well for situations where the population size is large and when it is hard to define the strength of an individual. The benefit with the tournament selection is that no global knowledge of the population is required. It does only strive to order and rank two individuals. The probability that an individual will be selected for next generation with tournament selection depends on:

- *The rank in the population*
- *Tournament size k*. The larger the tournament, the larger probability that it will contain high fitness members and the less probability that it will contain weak fitness members.
- *Probability p*. The fittest member of the tournament is selected with probability *p*. (p = 1 for deterministic, p < 1 for stochastic).
- *Individuals chosen with or without replacement*.

Research has proven that the expected time for a single individual with high fitness to take over the population is the same in tournament selection than in linear ranking selection. Tournament selection has been the most widely used selection method for GA applications due to its simplicity and ability to control the selection pressure by changing the tournament size *k*. (De Jong 2006: 128, Eiben & Smith 2007: 63-64)

Elitism

Elitism has a significant effect on the performance of a GA. Individuals with high fitness can be lost in selection or lose its good properties due to crossover and mutation. Therefore elitism is an additional selection method that always saves the fittest member in the population for next generation. At next selection iteration an individual with higher fitness than the previous elitism-individual can be replacing another individual with low fitness. (Mitchell 1998: 168)

3. ETHERNET

Ethernet is a computer networking technology for Local Area Networks (LAN) and is the most used LAN technology nowadays. Ethernet operates across two layers in the Open System Interconnection (OSI) model; the Data Link layer and the Physical layer as shown in Figure 7. The OSI model is a seven layer standardized method of describing communication between hardware and software in networks. A layer serves the layer above it and is served by the layer below it. The lower layers cover the standards that describe transmissions of data while the higher layers deal with reliability of data-transmission and presentation for the user. (Spurgeon 2000; Jaakohuhta 2005) In this case, concerning Ethernet, focus will be put on the two lowest levels in the OSI model.

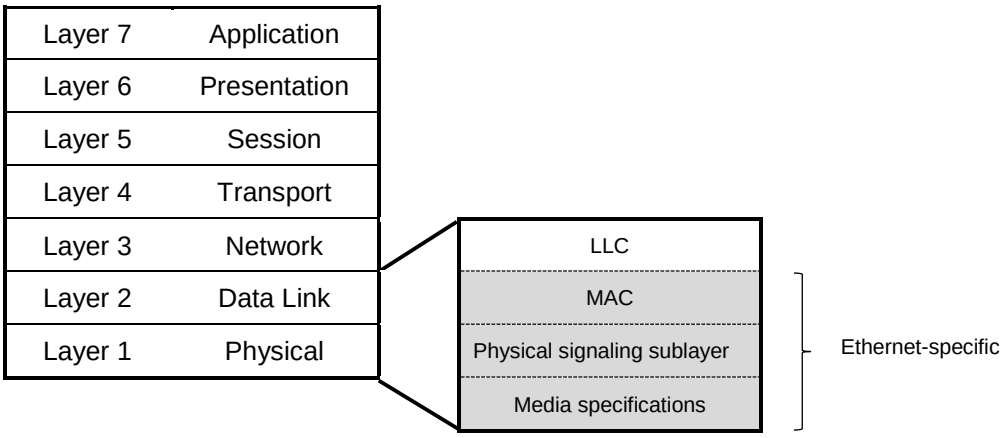


Figure 7. The OSI-model. (Adapted from Spurgeon 2000: 13)

3.1. History

The idea of Ethernet is based on the networking principles of the Aloha system from the late 1960s. The Aloha system was a radio based network for communication between the Hawaiian Islands. A station in the network was able to transmit data at any time, followed by a waiting time for acknowledgement. If an acknowledgement was not received within a certain time, the station assumed that another station had tried to transmit simultaneously, causing a collision. Avoiding another collision, the stations were distributed a random time for retransmission, giving both stations a higher probability rate of successful data transmission. (Jaakohuhta 2005: 9-31)

In 1973 Bob Metcalfe at the Xerox Palo Alto Research Center wrote a memo describing the Ethernet network system he had invented for interconnection between computer workstations and high-speed laser printers. The memo was based on the earlier Aloha system, but with a 100 % greater efficiency due to a collision detection mechanism. Along with a carrier sense mechanism the system was able to listen for activity on a channel with multiple stations, before transmitting. The Ethernet channel access protocol therefore got its name Carrier Sense Multiple Access with Collision Detect (CSMA/CD). (Spurgeon 2000: 3-22)

3.1.1. IEEE 802.3-standard

The original 10 Mbps Ethernet standard was first published in 1980 by DEC-Intel-Xerox, known as DIX Ethernet standard. Later on, the Institute of Electrical and Electronic Engineers (IEEE) wanted to develop the industrial Ethernet standard to a more office specific standard. In 1981 the IEEE defined the 802.3-standard which is the official worldwide Ethernet standard. The IEEE 802.3-standard covers communication speeds from 1Mbp/s up to the most recent 100Gbit/s with half-duplex, as well as full-duplex operation. Speed specific Media Independent Interfaces allow use of selected physical layer devices for operation over coaxial, twisted-pair or fiber optics cables. (Spurgeon 2000: 6-7; IEEE 2012)

3.2. The Ethernet frame

The Ethernet frame is a data package containing binary data. There are three main types of frames; unicast, multicast and broadcast. If the Least Significant Bit (LSB) of the most significant octet in the address is set to zero, the frame is a unicast-frame. Unicast-frames have an independent source- and destination address and are transmitted only between the sender and receiver. Contrary, if the LSB is a one, the frame is a multicast-frame and is transmitted between a sender and multiple receivers. The broadcast-frame, has a hexadecimal address of 0xFF for each byte. Meaning all bits are ones and the frame is transmitted to the whole network. (Jaakohuhta 2005: 83)

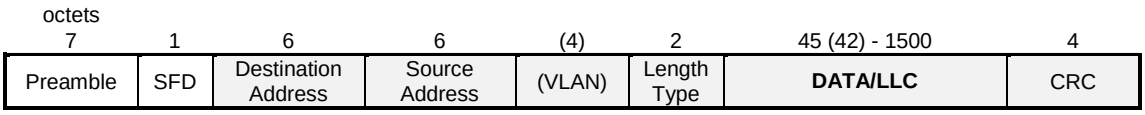


Figure 8. Ethernet IEEE 802.3 frame.

The structure of the IEEE 802.3 frame is seen in Figure 8. The frame is represented by octets, each octet containing 8-bits. The front of the frame begins with a 56-bit preamble, containing seven octets of the form 0101 0101. Due to signal start-up delays on the channel, the frame might lose a few bits in the beginning and therefore the preamble acts as a protector for the rest of the frame. The preamble is followed by a Start of Frame Delimiter (SFD), containing one octet of the form 1010 1011. (IEEE 2012)

The destination and source addresses contain six octets each. Each Ethernet station is assigned a unique 48-bit physical/hardware address, also known as the Media Access Control-address (MAC). Every station connected to the Ethernet channel reads each transmitted frame, at least as far as the destination address field. If the destination address does not match the station address the rest of the frame is ignored. The source address is the physical address of the station that sent the frame. The source address is not interpreted by the Ethernet MAC protocol but is provided for the use of high-level protocols, such as TCP/IP.

The optional four octet Virtual LAN (VLAN) tag header described by the 802.1Q standard is used for VLAN identification. When using managed switches the VLAN tag header directs the Ethernet traffic to specific ports of a switch that are assigned to a given VLAN. An addition of the VLAN tag header causes an extension of the original maximum frame size of 1518 bytes to a new maximum of 1522 bytes. However, the operation of the devices following the previous standard is not disturbed by the VLAN tag header since the first two bytes of the header contain a valid Ethernet type identifier.

The length or type indicator is a two octet field describing the length or type of the following data field in the frame. Since 1997 the type field has also been included in the 802.3-standard. Previously the 802.3-standard only covered the length field while the type field was covered in the DIX-standard. Therefore, the value in this field will determine whether it represents length or type. The normal length of the data field is minimum 46 bytes and maximum 1500 bytes. The value in length/type field has to be equal to or less than the maximum frame size, excluding the preamble and SFD. If so, the value will indicate that a Logical Link Control (LLC) header, defined by the 802.2-standard, is included in the beginning of the data field. If the LLC header plus data is less than 46 bytes a padding will be added to fill the remaining required data length. In the other case, if the value in the length/type field is greater than or equal to 1536 ($0x0600_{16}$), then type is described as specified in the DIX-standard. The hexadecimal value identifies the protocol used in the data field (e.g. 0x0800 for IP). Using the DIX-standard the network protocol software is responsible for providing minimum required data length in the data field.

In the data field, if length is described, a LLC header is added as mentioned above. The LLC header is a three octet field containing Destination Service Access Point (DSAP) for high-level protocol identification, Source SAP (SSAP) and a control byte. (Jaakohuhta 2005: 86; Spurgeon 2000: 42-45, 73-74)

Finally, the last field in the frame is the frame check sequence, also called Cyclic Redundancy Check (CRC). This 32-bit field holds a value for checking the integrity of the various bits in the frame fields. The CRC value of the frame is calculated with a known

polynomial, both by the transmitter and by the receiver. The frame check sequence is then compared, and if the CRC's differ a transmission error has occurred and the frame is dropped.

Concerning frame transmission over Ethernet, the frames are sequentially transmitted in octets, each octet containing 8-bits, with the most significant octet first transmitted on the channel. However, within the octets, the LSB is transmitted first. The principles are shown in the table below. (Spurgeon 2000: 42-46; Jaakohuhta 2005: 87)

Table 3. Frame transmission on Ethernet. The leftmost bit transmitted first.

Dec	240	46	21	108	119	155
Hex	0xF0	0x2E	0x15	0x6C	0x77	0x9B
Bin	**1111 0000**	**0010 1110**	**0000 0101**	**0110 1100**	**0111 0111**	**1001 1011**
←	**0000 1111**	**0111 0100**	**1010 0000**	**0011 0110**	**1110 1110**	**1101 1001**

3.3. Media Access Control Protocol

Ethernet is based on the CSMA/CD Media Access Control (MAC) protocol in half-duplex mode, where half-duplex refers to transmission in one direction at a time. This protocol determines when the Ethernet stations are allowed access to the channel and what action to take when a collision occurs. The condition when a signal is transmitted on the Ethernet channel is known as a carrier. Carrier Sense with Multiple Access therefore means that every interface in the network must wait for idle channel before transmission and each interface has the same priority in order of transmission.

Full-duplex operation means communication in both directions, and the CSMA/CD protocol is excluded. Instead it uses optional MAC control and PAUSE mechanisms for flow control. To ensure that both ends of a link operate at full-duplex mode an Ethernet

Auto-Negotiation is automatically initialized in the 802.3 standard. However, Auto-Negotiation is not supported in all Ethernet media types. E.g. fiber optics has its own auto-configuration setup. Therefore, it might in some cases be necessary to manually configure all end links to full-duplex mode.

The MAC control protocol in full-duplex mode is identified with the type value 0x8808, and indicates that a flow control is used to control when Ethernet frames are sent. The PAUSE operation is carried in the data field of the MAC protocol frame to define the time period the receiving station is requested to halt the transmission of data. (Spurgeon 2000: 76-84). This is however away from the limits of this thesis and is not taken into deeper discussion.

3.4. TCP/IP protocol

Deeper insight in the TCP/IP protocol will not be discussed in this thesis, but the main functions about the protocol is mentioned next.

Payload between computers in a network is carried in the data field of the Ethernet frame and structured as high-level network protocols. The protocol information carried in the frame establishes communication between computers attached to the network. The most common high-level network protocol is the Transmission Control Protocol/Internet Protocol (TCP/IP), which is divided into four layers.

The lowest link layer includes the device driver in the operating system and the corresponding network interface card in the computer. The following network layer deals with the movement of data within the network. Every device in a wide area network (WAN) has a unique Internet Protocol (IP) address of 32-bit numbers. The addresses are written as four decimal numbers, one for each byte, called dotted-decimal notation. The next transport layer provides a reliable flow of data between two hosts. Two applications using TCP must establish a connection before data exchange. The highest layer,

the application layer, finally handles the details of the particular application. The interaction between the IP and TCP protocol can be seen from figure 9. (Stevens 1994: 2, 7)

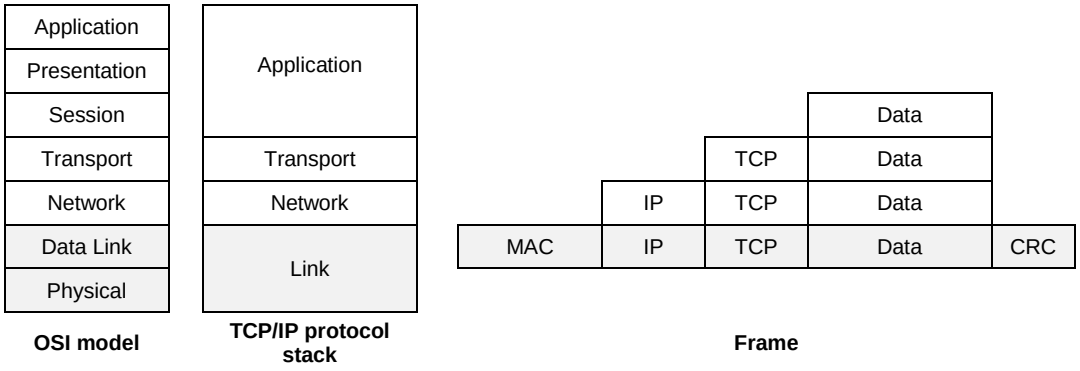


Figure 9. The TCP/IP protocol compared to the OSI model and the frame structure. (Reference from Stevens 1994)

3.5. IEC 61850 standard

In 1994 the International Electrotechnical Commission (IEC) presented a standardization of communication in substation automation systems (SAS), called the IEC 61850 standard. The objective of the standardization was to bring compatibility between different kinds of intelligent electronic devices (IEDs), regardless of the manufacturer. A SAS typically consists of several IEDs, which can be protection relays, circuit breakers and controllers. The communication between the IEDs is defined by the Generic Object Oriented Substation Events (GOOSE) protocol. (IEC 61850-1: 9-11, Söderbacka 2013: 11-13)

The hierarchy of substation automation can be seen from figure 10, where the substation is divided into three different levels such as station, bay and process level. At process level, data and status information is collected from primary equipment such as transformers or circuit breakers, but also operational functions such as tripping of circuit breakers and control of disconnecting switches is possible.

Secondary equipment such as control- and protection devices is located in the middle level, or the bay level. Typically, in medium voltage bays the control and protection functions are located within the same IED, while in a high voltage bay the control and protection functions are usually split into different IED's.

The station level is the supervisory level where the operator computers are located. The station – bay level communication, and the bay – process level communication is usually divided as two physically separate networks. It is however possible to share both station bus and process bus on the same network using IEC 61850 communication. (Söderbacka 2013: 15-16)

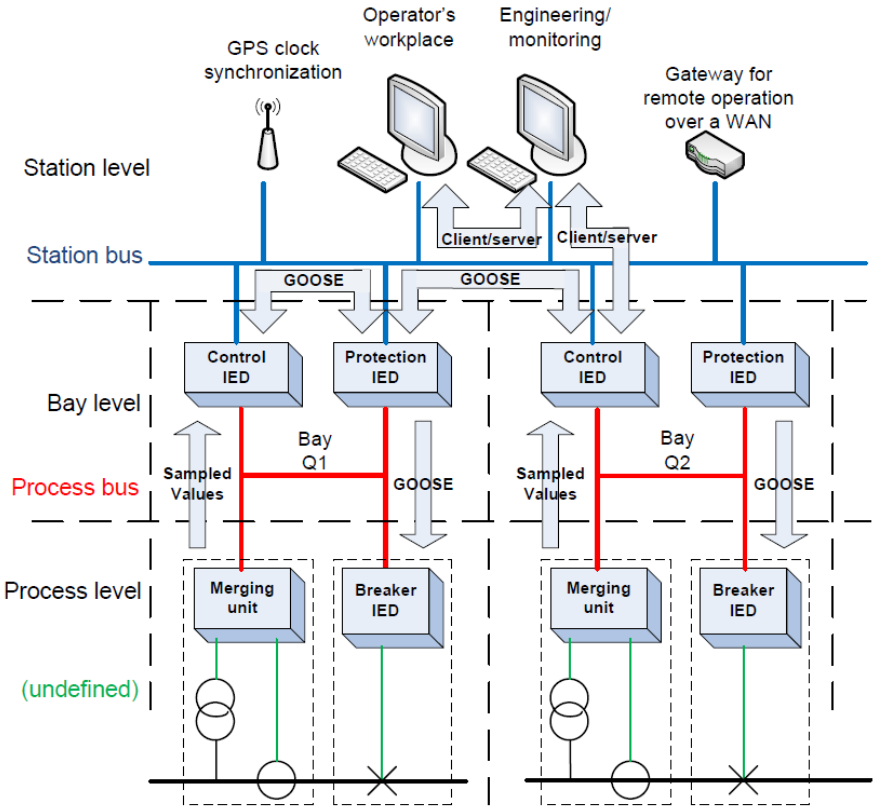


Figure 10. Distributed protection system of an electrical substation using IEC 61850. IEDs communicate mutually with GOOSE messages over the station bus, which is the central part in this project work (Söderbacka 2013: 15)

3.5.1. GOOSE protocol

The GOOSE protocol allows broadcasting of multicast messages across the LAN and is covered within three layers in the OSI-model, the physical-, data-link- and the application layer. Since it would need a thorough review to describe the entire GOOSE frame, which is out of scope in this thesis, only the structure is illustrated in figure 11. (Kriger et al.: 2013)

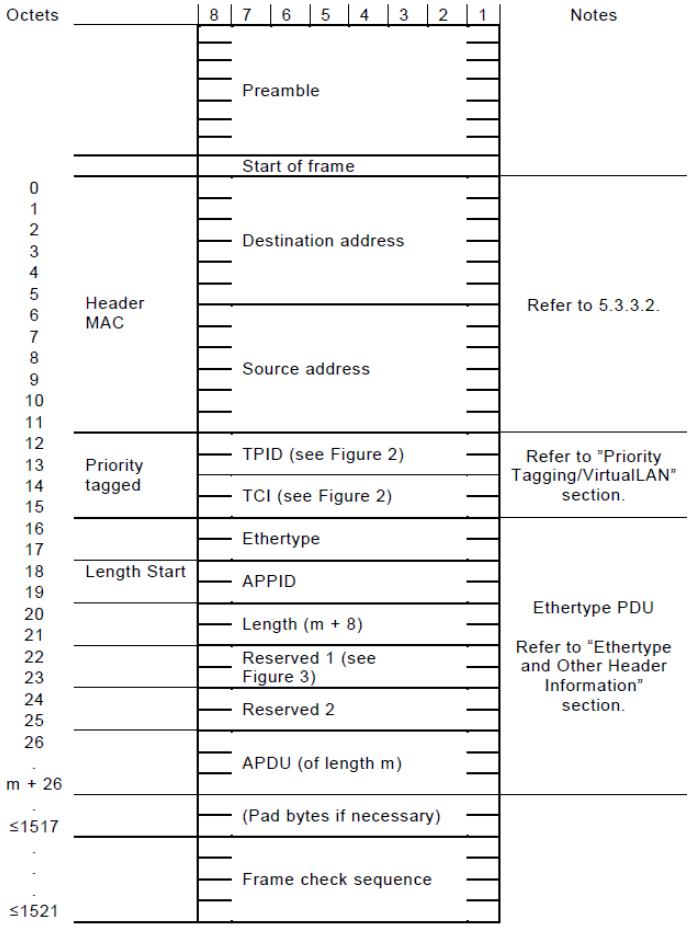


Figure 11. Frame structure of the GOOSE (ISO/IEC 8802-3) protocol. (IEC 61850-9-2: 27)

4. FIELD PROGRAMMABLE GATE ARRAY

The request of smaller, faster, cheaper and more complex circuit designs has pioneered the development of the Field Programmable Gate Array (FPGA). An FPGA is a hardware-configured programmable logic device (PLD) that can be reprogrammed. Compared to other software programmed integrated circuits, such as microprocessors, microcontrollers and digital signal processors, the FPGA is configured using a hardware description language (HDL). Software solutions can be relatively slow in many cases and this is why hardware design plays a central role in speed and parallelism. The benefit of using FPGAs instead of Application Specific Integrated Circuits (ASIC) lies in cost in small quantity productions and the ability to easily re-design a prototype. (Ashenden 2008: 30)

The core of an FPGA is formed by "a regular array of basic programmable logic cells (LC) and a programmable interconnect matrix surrounding the logic cells". The core is further surrounded by programmable I/O cells and the programmable interconnect is placed in routing channels. (Grout 2008: 28)

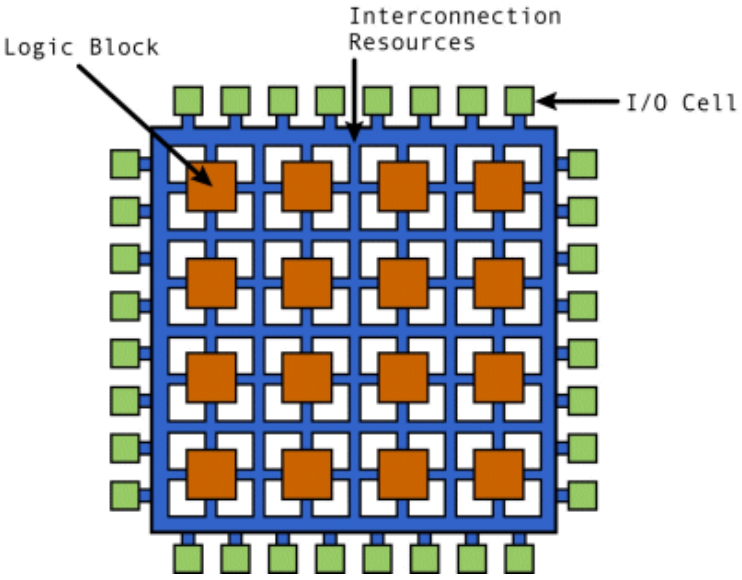


Figure 12. Generic FPGA architecture. (Zeidman 2006)

4.1. Altera DE4 board

The hardware platform used in this project is the Altera DE4 board with a *Stratix IV EP4SGX230KF40* processor. The processor features i.a. 228 000 logical elements, 17 133 Kbits of memory and 744 user pins (I/O). The platform is i.a. equipped with four Gigabit Ethernet RJ-45 ports and a RS-232 port specifically needed in this project. The programming connection between the host computer and the board is taken care of via a USB blaster port. (Altera 2012b)

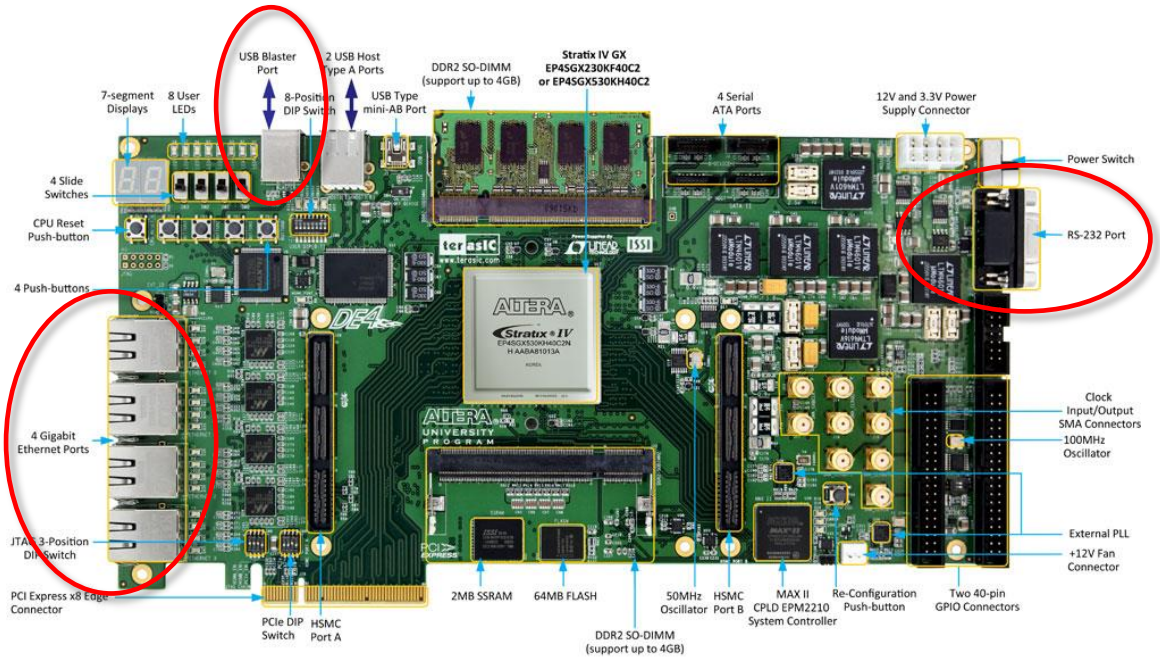


Figure 13. Altera DE4 board. Specific ports used in the project are circled. (Altera 2012b)

4.2. Hardware Description Language

Hardware Description Language (HDL) design is based on "the creation and use of textural based descriptions of a digital logic circuit or system". HDL design is divided into three different levels of abstraction; design specification, register transfer level and logic gates. The design specification level describes the behaviour in form of algorithms defined by architecture. Simply put, the architecture defines how the functional blocks of the algorithms are connected. The second level, the register transfer level (RTL), de-

scribes the flow of data, storage, and logical operations performed on the data. The design structure of a RTL is expressed in terms of logic gates and the wiring between them. Finally, at the lowest level, the logic gates are implemented as transistors. The most common HDLs in use today are VHDL and Verilog (Grout 2008: 193)

4.3. Verilog

Verilog was introduced in 1983 by Gateway Design System Corporation and later in the 90's it became an IEEE standard. The differences between Verilog and VHDL are many. Verilog is closely reminding of C-language and is said to be simpler and closer to hardware than VHDL. According to Zwolinski, "Verilog can be used to model logic circuits at the transistor or switch level, which is difficult in VHDL". Furthermore Verilog is compatible with fault simulators, which is not the case for VHDL. On the other hand, the benefit with VHDL is that it is better suited for behavioural modelling than Verilog because of its high-level constructs and abstract data types. (Zwoliski 2004: 327, Grout 2008: 196)

The choice of selecting Verilog as the programming language for this project was mainly that it reminds more of C than VHDL and because the system is to be compatible with the *EtherTester* programmed by Olli Rauhala in the Teho-FPGA project. 4

```
module fullAdder (A,B,Cin,S,Cout);

   input A,B,Cin;
   output S,Cout;

   wire X1,X2,X3;

   xor (X1,A,B);
   xor (S,X1,Cin);
   and (X2,X1,Cin);
   and (X3,A,B);
   or  (Cout,X2,X3);

endmodule
```

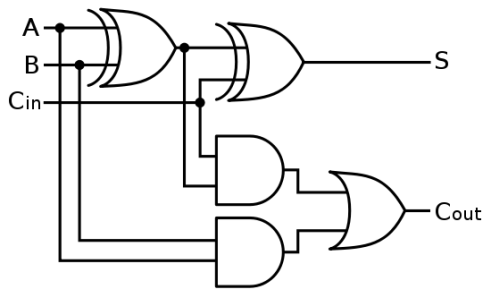


Figure 14. A simple example of a full-adder implemented in Verilog with RTL-view to the right.

5. TESTER IMPLEMENTATION

The main idea of this project was to generate and modify the payload of Ethernet frames with Genetic Algorithms on a FPGA hardware implementation. By constantly sending a flood of different frames to an IED, will there be any frame that the IEC 61850 implementation cannot interpret and cause the IED to crash?

5.1. Hardware implementation

The generation and modification of the payload is done within the hardware and programmed in Altera Quartus II with Verilog HDL. The hardware consists of a Frame Generator module, a Transmission module and the brain of the system; the Genetic Algorithm module. A random initial population is created simultaneously with the Frame Generator, where each 32-bit randomly generated part is stored into larger registers in the Transmission module. When the total initial population is created, the individuals are transmitted sequentially over the network, while a random 32 bit part of each individual is chosen as seed for GA modifications. Depending on how well the individuals affect the DUT (in a bad sense), feedback is received by the system and the chosen parts from the individuals are sorted in descending order according to the fitness values. Individuals with high, medium and weak fitness will then be combined as crossover pairs and the offspring from each parent mutated to a resulting children. When the GA operations are ready the modified parts are thrown back into the same positions in the initial population, and transmitted over to the network again. New positions in the population are again selected randomly and used as next seed, and the whole procedure continues again (see Figure 15). The GA hardware is combined as a part of the EtherTester hardware developed by Olli Rauhala, and interacts with the framestormer submodule.

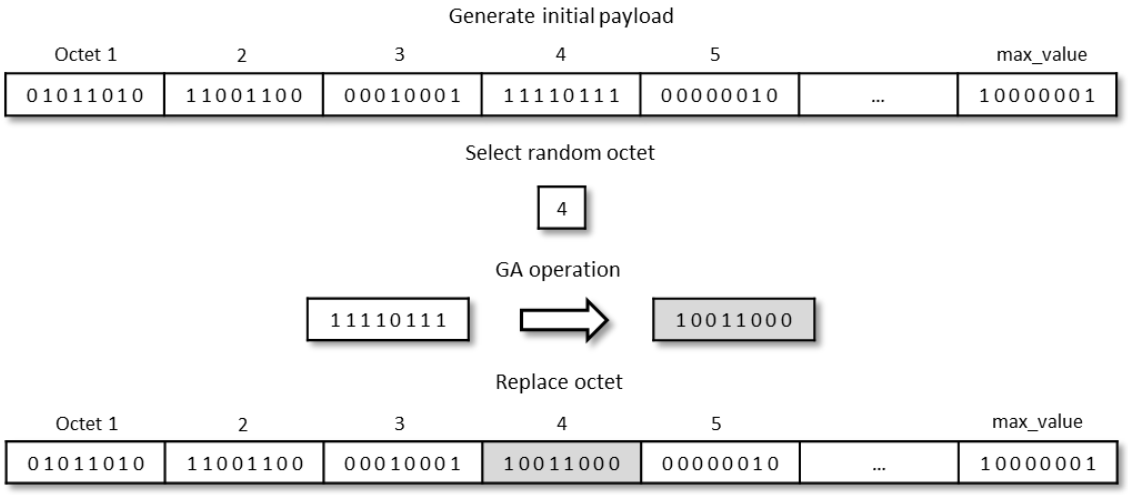


Figure 15. Initial payload generated and saved to a register with maximum length defined by the user. A randomly selected octet is chosen for GA modification. New octet is replaced and saved to same position in array register. An 8-bit representation is used in this figure for a clearer presentation. Same procedure holds for every population, but with different octet replacing indexes.

Next follows a deeper presentation about separate modules, while at the end of the chapter there is a description of the complete system.

Frame Generation with Linear Feedback Shift Register

Compared to other programming languages, HDL do not support direct commands for random number generation and the random number generation must be implemented within the hardware. A common way to do that is with a binary Linear Feedback Shift Register (LFSR). At specific positions, also called taps, the bits are compared for equality by an exclusive-nor gate and the result connected as a feedback to the MSB in the register. At each iteration, the bits are shifted to the right, causing a random pattern generation until the register reaches 2^n-1 (n = number of bits) before it starts repeating. To illustrate this, an 8-bit LFSR is presented in Figure 16.

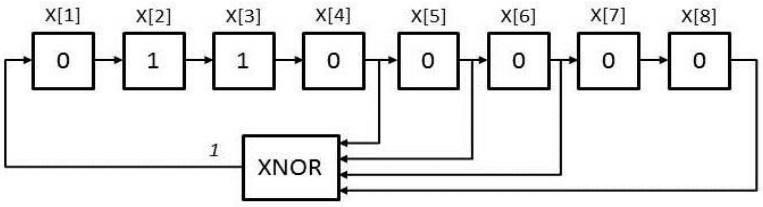


Figure 16. 8-bit LFSR with tap positions at x[8], x[6], x[5] and x[4]. Feedback from XNOR gate is shifted to the MSB x[1]. The 8-bit register has 2^8-1 = 255 states until it starts repeating. Tap positions for maximum randomness with 255 states is applied from Xilinx. (Ciletti 2011: 171-179, Xilinx 1996: 5)

In this project a 32-bit LFSR was implemented in Verilog, since 32-bit is maximum bit length for an output on the Altera DE4 processor. Tap positions are attached to x[32], x[22], x[2], x[1] and the register has $4.3 \cdot 10^9$ states before repeating. (Ciletti 2011: 171-179, Xilinx 1996: 5)

The LFSR module in the program requires five inputs; clock signal, system-enable, length, randomization command and an initial state. The local register in the module is initialized with an initial state of predefined zeros and ones. At system-enable (reset) the LFSR register is given the initial value and the register is put in running state. At each positive edge clock-pulse in running state, the bits in the local register are shifted and XNOR'd at specified tap positions. When the randomization command enters a high-state, the actual value in the local LFSR register is assigned to the output frame of the module, and a running signal is put into high state as long as a counter value is less than the set length. The Verilog code for the LFSR is given in Appendix 3.

For generating the random initial population, the LFSR module is accessed for each individual. With a population size of *n* individuals, same amount of LFSR's are needed. Each individual is assigned a hardcoded initial value, which then is processed by the LFSR module as long as the user defines the generation of initial population to start. In this way the individuals will always receive sufficient randomness on each system run.

Fitness selection

The selection method for the program is based on the deterministic mechanism using tournament selection. In this way each individual will be selected once, meaning no individual is replaced or rejected, only sorted by means of fitness. Depending on the input feedbacks, the 32-bit frames are sorted with the bubble-sort algorithm in descending order. The two frames with highest fitness will be combined as first crossover pair. Similarly the second best pair is combined as crossover pair 2, and the weakest pair combined as crossover pair 3 (see Appendix 2). When all frames are sorted a ready impulse is sent to the next crossover module. See Appendix 3 for full hardware implementation of the Fitness selection module.

Crossover

As for the initial population a random crossover index vector is generated in the crossover module, based on same LFSR used earlier. To get varying crossover indexes for each crossover module, the LFSR registers are set with predefined hardcoded initial values. At system enable, each LFSR register in the crossover module will take the initial state value defined, and put into running mode. Always when the crossover module receives a positive-edge ready from the previous fitness module, the actual value of the shift register is written to output and an enable signal is sent to the mutation module.

Instead of performing the crossover operations with a for-loop, logical expressions were used to save some time and because they are really easy to implement with only two lines of code (see Appendix 3). Crossover between two parents is determined by the randomly generated crossover vector which has a fixed probability rate of 0.5 due to the binary representation. Calculating the offsprings follows the uniform crossover with pseudo-codes:

$$O_1 = (P_1 \text{ and } (\text{not } C)) \text{ or } (P_2 \text{ and } C) \quad (3)$$

$$O_2 = (P_2 \text{ and } (\text{not } C)) \text{ or } (P_1 \text{ and } C) \quad (4)$$

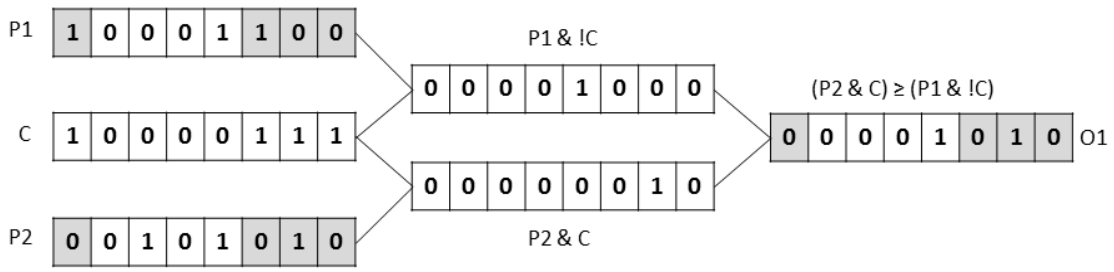


Figure 17. Principle of logical crossover between parents. The resulting offspring from the first parent (P_1) is presented in this example. Corresponding O_2 is created according to pseudo-code (4).

Mutation

Mutation within an offspring is supposed to happen with a very small probability. Having a frame of 32-bits and with the typical probability rate 0.05 will result in 1.6 % of having a binary 1 somewhere in the vector. That represents only half a bit so the mutation rate is chosen to a fixed rate $\frac{1}{32}$, which equals 3.1 %. The initialization of a mutation index vector is implemented by a decoder. A decoder has *m* inputs and *n* outputs, with $n = 2^m$. Meaning that a 32-bit wide output frame demands a 5-bit wide input. By generating a random number with a 5-bit LFSR, the random number will represent a 32-bit wide output frame where only one bit is set to 1. This will not cause too large mutation within the offspring, but exactly one bit will always be mutated. (Ciletti 2011: 62-63)

As for the crossover module, the LFSR for random number generation in the mutation module is set to an initial state and starts shifting at system enable. When the mutation module receives an enable command from the crossover module, the mutation index vector is set according to the actual LFSR value. The mutation operation is then calculated and a trigger pulse is confirming that GA is done.

For binary representation the mutation operation is quite simple. To generate the resulting children the offspring is compared to the mutation index vector with a XOR-gate, forcing a 0 to 1 and 1 to 0, if the mutation index contains a one at that position (see Appendix 3).

Frame transmission

The interface between the GA hardware and the framestormer is implemented with a 32-bit data output and an enable output that tells the framestormer when a new frame is ready to be sent, and for how long. Additionally a ready signal (*FS_ready*) from the framestormer is used for frame selection. Always when a new generation is ready to be transmitted, the module is put into running mode and a counter for frame selection is initialized. In transmission mode, the selected frame is set as output until the clock pulses equals the set *length* value. In that way the framestormer will transmit the 32-bit frame as a payload as long as the *enable_framestormer* signal stays high. For full payload size (1600 bytes) the clock pulse counter has to be set to 400 positive edge counts. When the framestormer has transmitted the entire package, the ready signal is sent to the *GeneratePayload* module (see Appendix 1) and the next frame is selected. When the last package is transmitted and all the fitness values are received by the fitness module, an enable fitness signal will trigger the fitness module and the process is repeated again.

System structure – Top level design

The GA hardware requires eleven input signals and two output signals:

- *Input clock*
- *Input enable_system*
- *Input randomize*
- *Input [8:0] length*
- *6 input [31:0] feedback*
- *Input framestormer_ready*
- *Output enable_framestormer*
- *Output [31:0] out_data*

Enable_system (reset) is a positive edge triggered input signal that sets all shift registers with the initial hardcoded values and commands the shifting to start. Because of the

parallelism, with six individuals modified simultaneously, initial values are needed for retrieving sufficient randomness. This signal is triggered by the user at start-up.

Randomize is a positive edge triggered input signal that ensures the randomness for each system start-up. When this signal enters a high-state the generation of the initial population starts. This signal is randomly triggered by the user, after *enable_system*.

Length is a 9-bit input value entered by the user. The length value tells the LFSRs how long to generate the initial population and defines the maximum length of the registers in the Transmission module. Since the LFSRs generate 32-bit frames on every clock pulse, and the *out_data* is 32-bit wide, the maximum size of the length parameter is 400 for a 1600 byte payload. *Length* has to be set before *enable_system* and *randomize*.

Feedback inputs are controlled by the software running in the NIOS processor. The values received from the Raspberry Pi via RS-232 connection are 32-bit unsigned integers. When all feedback values are received from a generation the fitness selection is enabled and new members generated.

Input *framestormer_ready* is a positive edge triggered signal from the framestormer module which confirms that frames have been transmitted and that the framestormer is idle. Additionally the signal determines the value of a frame counter, allowing a new individual to be transmitted.

Output *enable_framestormer* is an output signal that stays in high state as long as the framestormer is sending a 32-bit part of an individual on *out_data*.

The complete system structure is described in Appendix 1.

5.2. System simulation

Simulation of separate modules and the complete system is done with the Altera U.P. Simulator with Vector Waveform File generation (VWF). In the VWF, inputs, outputs and registers can be inserted as nodes, where the inputs can be modified to simulate the

program behaviour. Next follows a few simulation results from parts of the system modules. Notice that the simulation results for the frames is grouped as bytes (8 bits) for a clearer illustration.

Frame Generator module with LFSR

When the user commands the *enableSys* to go high, the *LFSR*s are assigned with initial states and start running. At user command *randomize*, the frame outputs are assigned with the LFSR values, and changes at each clock pulse according to the set *length* value. The *length* input, in this simulation case a 4-bit value, defines the total width of each individual in the initial population. The *FG_run* output works as a payload-size indicator to the *GeneratePayload* module (see Appendix 1), and will stay high as many clock pulses as defined by *length*.

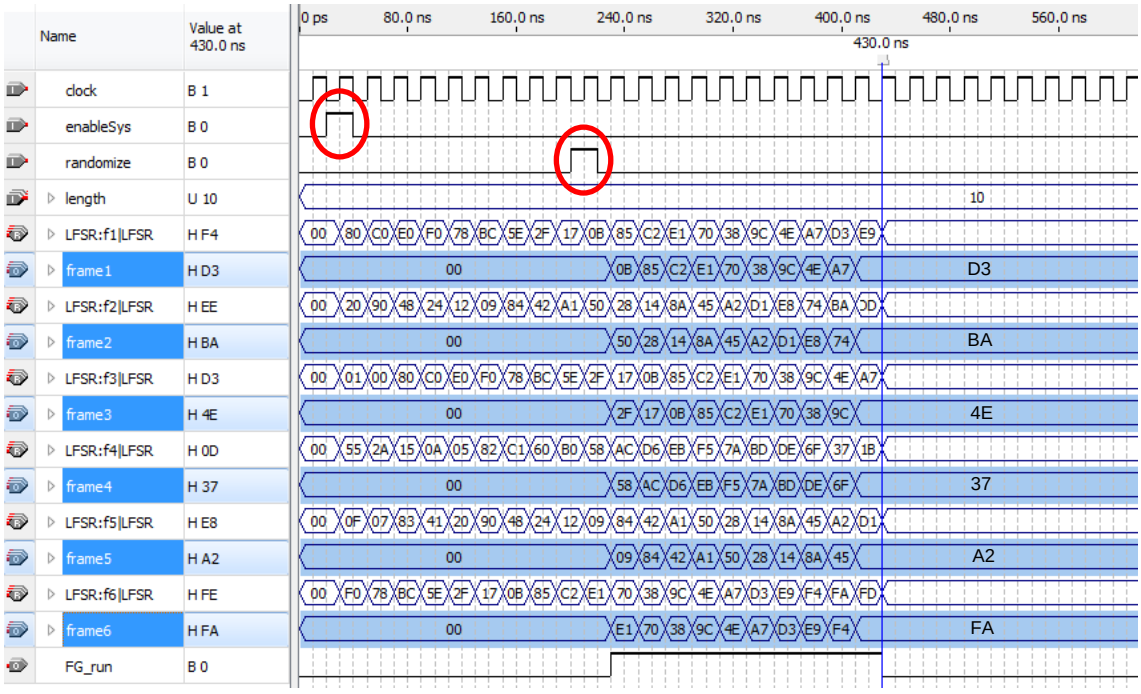


Figure 18. LFSR simulation. At *enableSys*, *LFSR*s are assigned with initial states and starts shifting. At user command *randomize*, frame outputs are assigned with actual LFSR values and *FG_run* stays high as many clock pulses according to value *length*. With length 10 the initial population for e.g. frame 1 will get the hexadecimal value of 0x0B:85:C2:E1:70:38:9C:4E:A7:D3.

GA module

When *enableSys* is received, the LFSRs for crossover index vectors and mutation index vectors are set to predefined initial states. When the *enableCO* signal, triggered by the fitness module, goes high, the crossover index vector takes the actual LFSR value and crossover is applied to the parents, resulting in new offsprings. A ready signal is sent to the mutation module and the actual value of the LFSR is decoded to a mutation index vector. Mutation is then applied to the offsprings and a GA_ready signal is confirming that new children have been generated.

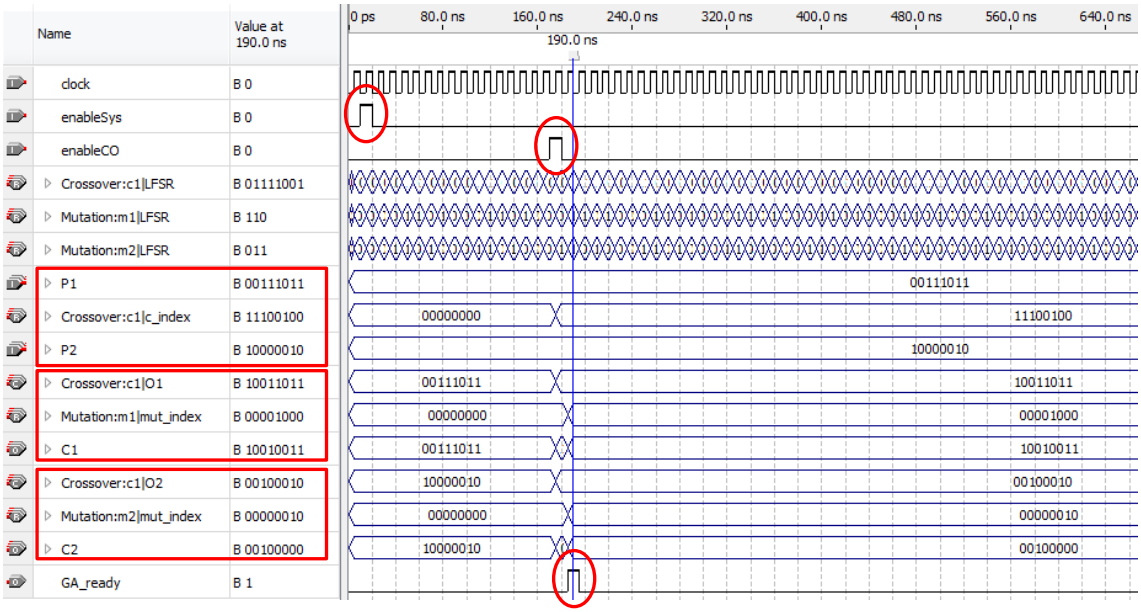


Figure 19. GA simulation for two input parents with two output children. On system start-up (*enableSys*) the LFSRs are assigned with initial states and starts shifting. When an *enableCO* command is received from the fitness module, crossover and mutation is applied to the partents and *GA_ready* is confirming when operations are done. *Uppermost box*: Parents *P1* and *P2* crossed according to *c_index*. *Middle box*: Offspring *O1* mutated with *mut_index*, resulting in children *C1*. *Lowest box*: *O2* mutated with *mut_index*, resulting in children *C2*.

Fitness module

Inputs to the fitness selection module are randomly selected parts from each individual in the population (see Figure 15 in chapter 5.1). The feedbacks are 32-bit unsigned integer values describing how the complete frame did affect the latency of the communication. The feedbacks are sorted in descending order, along with the frames they belong to. In this simulation test, frame *I6* has caused the greatest latency (*FB6* = 13) and is therefore sorted to output *S1*. Frame *I1* has the second greatest latency and sorted to output *S2*, and so on. When all the frames are sorted, a ready signal is triggered and it will enable the next crossover module.

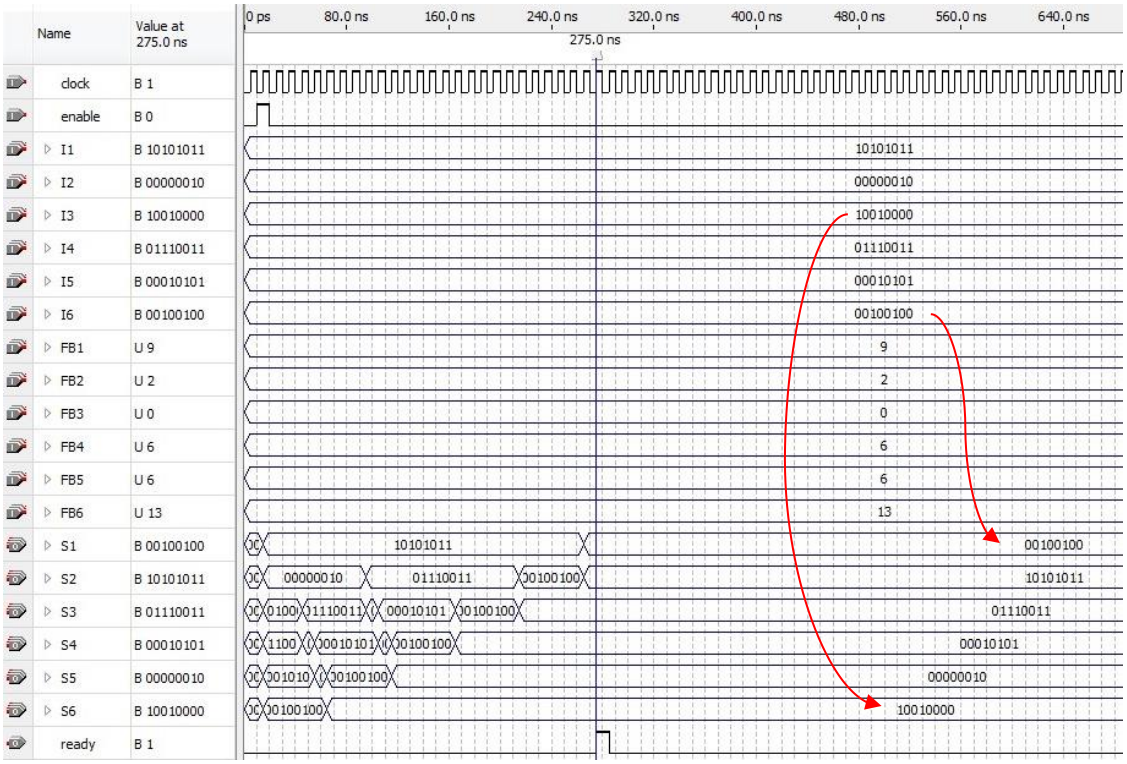


Figure 20. Fitness simulation. Frame sorting in descending order with Bubble-Sort algorithm. Frame *I1* is compared to fitness *FB1*, *I2* to *FB2* and so on. Frame *I6* has the highest fitness (*FB6* = 13) and is sorted to output *S1*. The opposite smallest feedback belongs to *I3* (*FB3* = 0), and therefore sorted to output *S6* (see arrows). At timeline 275 ns the sorting is completed and confirmed by a ready signal.

5.3. EtherTester and GA interface

The GA generator in this project is interfacing with the FPGA EtherTester, programmed by researcher Olli Rauhala at the University of Vaasa. The EtherTester is implemented on the Altera DE4 using Triple-Speed Ethernet (TSE) with 100Mbit/s bitrate. Altera's Triple-Speed Ethernet is available with 10/100/1000 Mbit/s network connections. The system consists of a "framestormer" that sends a flood of Ethernet frames and a "frame-analyzer" that receives and counts the incoming Ethernet frames. Transmission and reception is executed within the hardware using the AVALON Streaming interface, while the NIOS II processor controls the hardware accelerators over the AVALON Memory Mapped interface. The user interface is held on a host-PC, connected with RS-232 to the Altera DE4 platform. The system structure of the EtherTester is built with the Qsys system integration tool (see Figure 22).

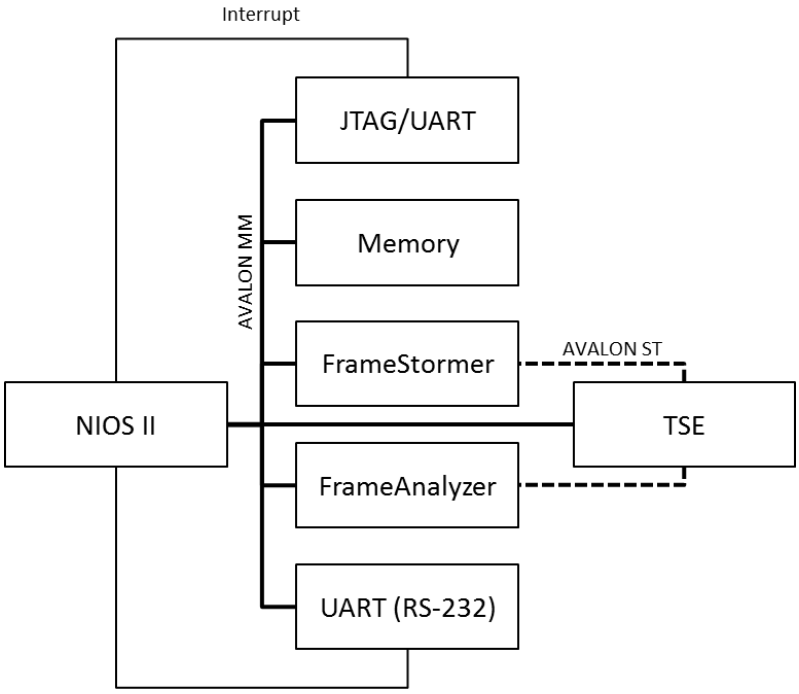


Figure 21. System structure of EtherTester in Qsys. The NIOS II software controls and monitors the hardware accelerators by the Avalon Memory Mapped interface. FrameStormer and FrameAnalyzer interfaces with the TSE by Avalon Streaming Interface.

With the purpose of only transmitting random data to a DUT, there is only need for linking the GA hardware with the framestormer. Using Avalon Memory Mapped connection, controlled by the NIOS software, the user is able to start the randomization- and the generation process by entering specified commands from the keyboard. The NIOS software also controls the feedback inputs to the GA hardware. After receiving as many feedbacks that equals the population size, the sorting and GA operations are enabled. The interface between the GA and the framestormer system is integrated with Qsys. The enable-framestormer signal and frame data beween the systems are connected via conduit connection, while the rest of the inputs to the GA system are connected with Avalon Memory Mapped connection.

5.4. Test environment

The test environment consists of a DE4 Altera FPGA, sending Ethernet frames via a switch to a DUT (Device-Under-Test), in this case a protection relay. The DUT communicates over Ethernet within the IEC61850 standard, using the GOOSE protocol. The Ethernet frames that are generated with the FPGA are therefore constructed as GOOSE frames, only with the payload in the frame modified with genetic algorithms. Additionally, with assistance of Mika Ruohonen who has a wide knowledge about the GOOSE protocol and configurations of protection relays, an application of a Raspberry Pi single board computer were implemented for feedback calculations. The Raspberry Pi communicates with the DUT by GOOSE structured messages via the same switch, and measures the latency of the DUT while the FPGA is transmitting data. In the Raspberry Pi application a parameter, *packet exchange per frame*, is determining how many times the GOOSE message is sent between the Raspberry Pi and the DUT, until the latency value is measured and the FPGA is enabled to transmit a new frame. The observed latencies are sent as feedback values to the EtherTester via a RS-232 connection. Feedback is sent as three separate 32-bit unsigned integer values, including time spent in exchanging messages, number of missed dataset state changes, and number of missed or dropped messages. However, only the time spent in exchanging messages is used as feedback value to the hardware inputs.

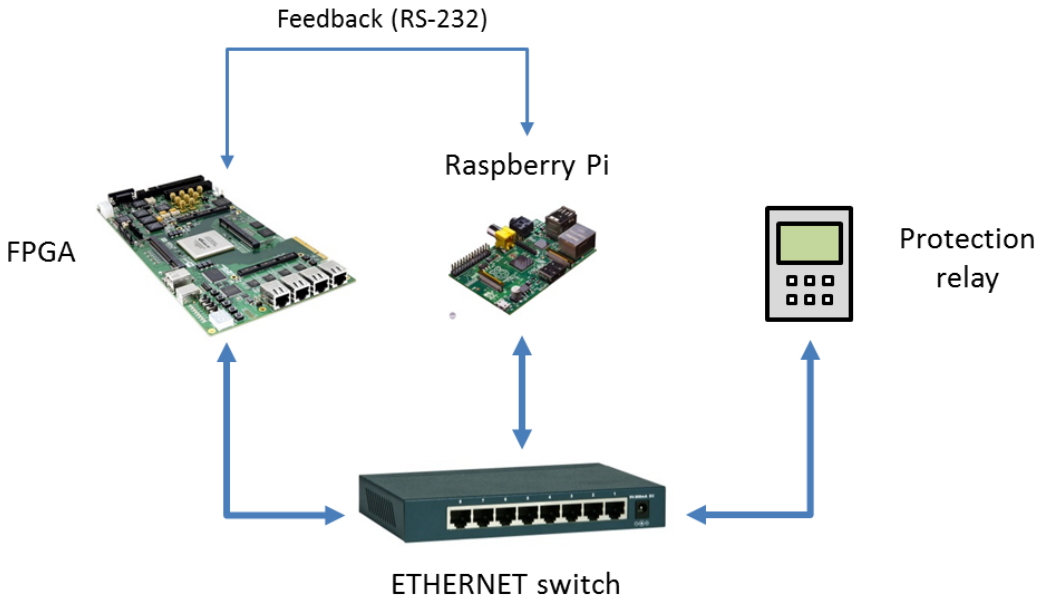


Figure 22. Overview of the test environment. The FPGA, Raspberry Pi and the protection relay (DUT) are connected to an Ethernet switch. The time-delay feedback is transmitted from the Raspberry Pi to the FPGA via RS-232 connection.

5.5. Field tests

Field tests were run in the DEMVE laboratory using a protection relay. The FPGA, Raspberry Pi and the protection relay were connected to an Ethernet switch according to Figure 22. Additionally, a monitor was attached to the Raspberry Pi for program control and monitoring. The control of the NIOS software in the FPGA is driven by the Altera Monitor Program. In the console application it is possible to set parameters and control the hardware system (detailed presentation in Appendix). Furthermore, enabling result tracing, the feedback values are continuously printed out to the Monitor console application and Ethernet traffic analysed by using the Wireshark network analysis tool. The results of the field tests will be presented next.

GA-hardware

The first functional test was to ensure that the properties of the GA hardware are working properly. By analysing transmitted frame-data with Wireshark it was confirmed that random parts of the frame changes from generation to generation. In Figure 23 it is presented how a single individual frame is changing from first- to second generation.

Frame 1 – First generation

```
⊞ Frame 195: 174 bytes on wire (1392 bits), 174 bytes captured (1392 bits) on interface 0
⊞ IEEE 802.3 Ethernet
⊞ Logical-Link Control
⊞ Data (156 bytes)    MAC address                 length

0000  ff ff ff ff ff ff 00 60  6e 11 02 0f 00 a0 c2 fb   .......` n.......
0010  a6 ad e1 7d d3 56 f0 be  e9 ab 78 5f 74 d5 3c 2f   ...}.V.. ..x_t.</
0020  ba 6a 9e 17 dd 35 4f 0b  ee 9a a7 85 f7 4d 53 c2   .j...5O. .....MS.
0030  fb a6 29 e1 7d d3 94 f0  be e9 4a 78 5f 74 a5 3c   ..).}... ..Jx_t.<
0040  2f ba d2 9e 17 dd e9 4f  0b ee f4 a7 85 f7 fa 53   /......O .......S
0050  c2 fb 7d 29 e1 7d be 94  f0 be 5f 4a 78 5f af a5   ..}).}.. .._Jx_..
0060  3c 2f 57 d2 9e 17 2b e9  4f 0b 95 f4 a7 85 4a fa   </W...+. O.....J.
0070  53 c2 25 7d 29 e1 12 be  94 f0 09 5f 4a 78 84 af   S.%})... ..._Jx..
0080  a5 3c c2 57 d2 9e e1 2b  e9 4f 70 95 f4 a7 38 4a   .<.W...+ .Op...8J
0090  fa 53 1c 25 7d 29 8e 12  be 94 c7 09 5f 4a 63 84   .S.%}).. ...._Jc.
00a0  af a5 31 c2 57 d2 18 e1  2b e9 0c 70 95 f4         ..1.W... +..p..
```

Frame 1 – Second generation

```
⊞ Frame 1104: 174 bytes on wire (1392 bits), 174 bytes captured (1392 bits) on interface 0
⊞ IEEE 802.3 Ethernet
⊞ Logical-Link Control
⊞ Data (156 bytes)    MAC address                 length

0000  ff ff ff ff ff ff 00 60  6e 11 02 0f 00 a0 c2 fb   .......` n.......
0010  a6 ad e1 7d d3 56 f0 be  e9 ab 78 5f 74 d5 3c 2f   ...}.V.. ..x_t.</
0020  ba 6a 9e 17 dd 35 4f 0b  ee 9a a7 85 f7 4d 53 c2   .j...5O. .....MS.
0030  fb a6 29 e1 7d d3 94 f0  be e9 4a 78 5f 74 a5 3c   ..).}... ..Jx_t.<
0040  2f ba d2 9e 17 dd e9 4f  0b ee f4 a7 85 f7 fa 53   /......O .......S
0050  c2 fb 7d 29 e1 7d be 94  f0 be 5f 4a 78 5f af a5   ..}).}.. .._Jx_..
0060  3c 2f 47 e2 d7 3f 2b e9  4f 0b 95 f4 a7 85 4a fa   </G..?+. O.....J.
0070  53 c2 25 7d 29 e1 12 be  94 f0 09 5f 4a 78 84 af   S.%})... ..._Jx..
0080  a5 3c c2 57 d2 9e e1 2b  e9 4f 70 95 f4 a7 38 4a   .<.W...+ .Op...8J
0090  fa 53 1c 25 7d 29 8e 12  be 94 c7 09 5f 4a 63 84   .S.%}).. ...._Jc.
00a0  af a5 31 c2 57 d2 18 e1  2b e9 0c 70 95 f4         ..1.W... +..p..
```

Figure 23. GA-hardware test with Wireshark. Random part of data changes from generation to generation. The first 12-bytes contain the MAC-address, where the destination address is set to broadcast. The MAC-address is followed by a 2-byte length field, while the remaining 160 byte is randomly generated data by the hardware. The red box is the 32-bit randomly selected part of the data that has been GA-modified.

Transmission test

It is possible to adjust the waiting time between the transmissions of frames with the ICBF parameter, *idle cycles between frames*. The packet exchange rate is a parameter in the Raspberry Pi application that determines how many times a GOOSE message is exchanged between the Raspberry Pi and the DUT before a feedback value is calculated. The feedback value from the Raspberry Pi therefore equals the total transmission time divided by the packet exchange rate.

It was further detected that below a boundary of around 1.5 million cycles, communication between Raspberry Pi and the DUT starts malfunctioning and finally stops, while the hardware continues to flood packages. From the following figures (Figure 24-26) one can clearly notice that the transmission rate jumps between certain levels regardless of the ICBF value or changing the packet exchange rate. The smallest communication latency was however obtained when ICBF = 2 million (see Figure 25). Why the communication latency is smaller with an ICBF of 2 million than with 2.5 million is incomprehensible. This phenomenon is probably caused by disturbances or related to the timing of the transmitted frames from the FPGA.

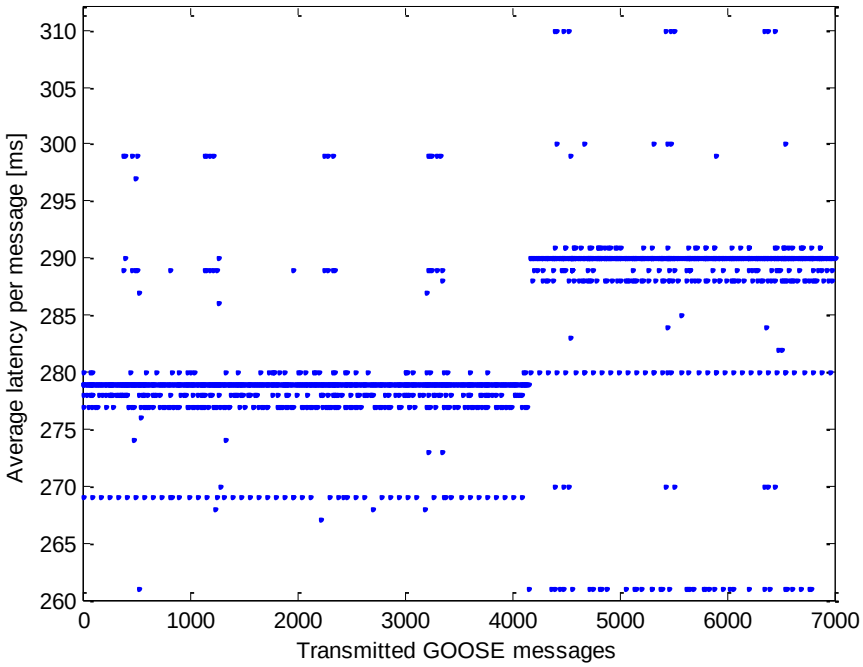


Figure 24. Communication latency when ICBF = 1.5 million and packet exchange rate = 10.

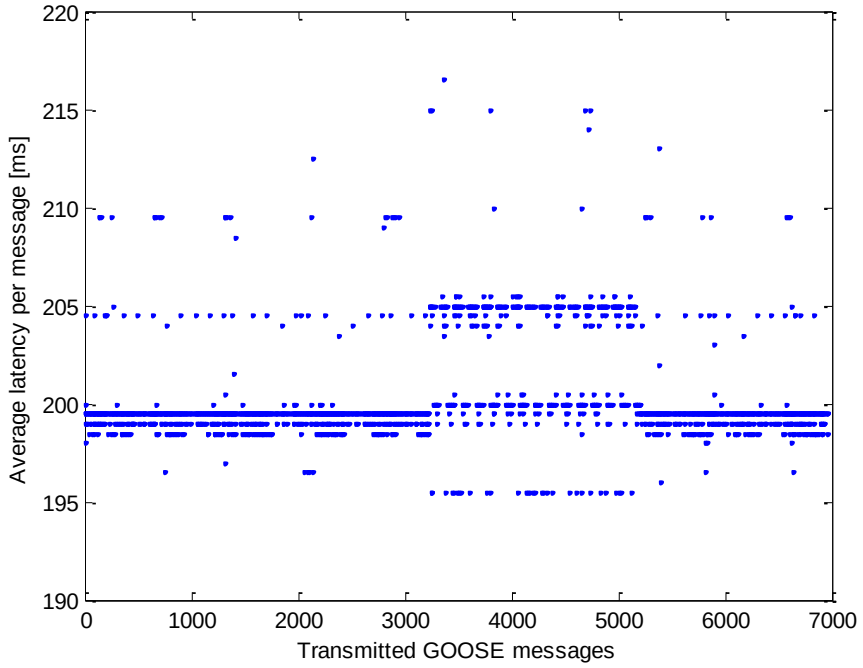


Figure 25. Communication latency when ICBF = 2 million and packet exchange rate = 10.

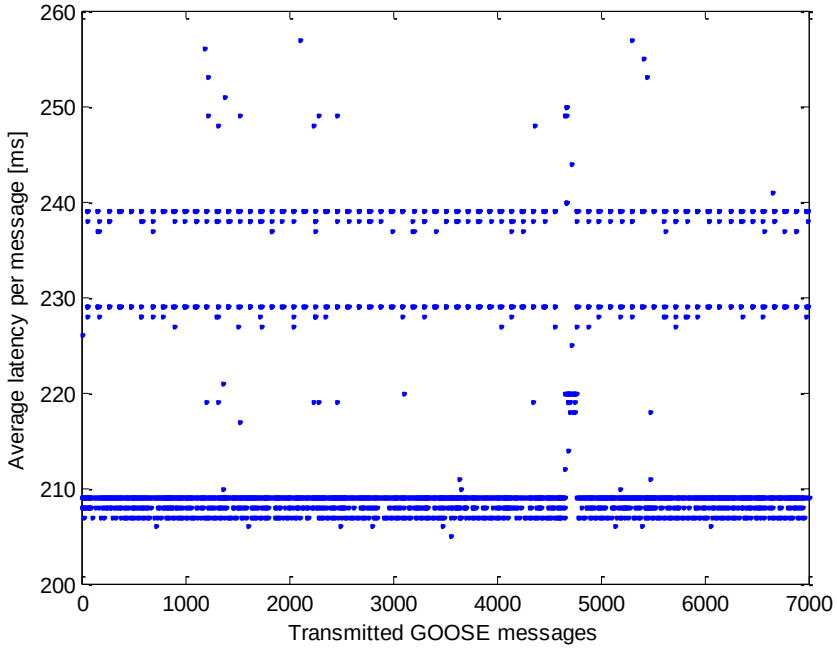


Figure 26. Communication latency when ICBF = 2.5 million and packet exchange rate = 10.

Safe mode test

In the same project group, researcher Mathias Björk did research in the same subject but with a GA-software application. Björk noticed that when the MAC-destination address is changed from broadcast to a non-existing address of only zeros, and with the ICBF parameter set at full transmission speed without delay, the DUT falls into an error mode and reboots for several times before it finally enters a safe mode. In safe mode, the DUT is inaccessible and totally out of service. The only way of rebooting the device is to manually disconnect the power supply and restart the device.

This same phenomenon was recreated with the hardware application. Occasionally, the DUT managed to reboot after a couple of errors, but usually it got stuck in the safe mode. With the ICBF parameter set to zero, there is not enough time for feedback calculations and for GA to generate new frames, so only the same population is transmitted continuously. However, it was shown that the safe mode only is affected by the destination address *0x00:00:00:00:00:00* and with full transmission rate.

6. CONCLUSIONS

In this thesis, the robustness of an intelligent electronic device (IED) was tested by implementing genetic algorithms in an FPGA hardware. By generating random Ethernet data and modifying the data with genetic algorithms on the FPGA, the hope was to generate harmful data for the IED. The complete system consists of an FPGA, a Raspberry Pi, and an IED, all connected to Ethernet via a switch. The FPGA continuously transmits GA modified Ethernet data to the IED, while the Raspberry Pi and the IED exchanges GOOSE messages mutually. Depending on latency of the IED affected by disturbing data, the latency time is sent as a feedback value from the Raspberry Pi to the FPGA via RS-232 connection. A higher feedback value describes a larger latency and is important in finding even better (in this case worse) frames.

The FPGA simultaneously generates a random population while the individuals are transmitted sequentially to the IED. The Raspberry Pi measures the latency caused by each frame and a feedback value is transmitted to the FPGA. When a single feedback value of a frame is obtained in the FPGA the next frame in order is transmitted over the channel. Finally, when the total population size is transmitted, the frames are sorted according to the feedback values, new members are generated and replacing the individuals of the old population.

The achieved results do not, however, confirm the impact of the genetic algorithm search method. According to the time-measurement plots in chapter 5.5, there is no sign of increasing transmission time that would be caused by the optimisation. Instead the transmission time jumps irregularly between certain levels, probably due to unknown delays caused by the Raspberry Pi, delays in the IED, or synchronization between Raspberry Pi and disturbing frames.

However, during the research project, it was observed that it is possible to make the IED to enter a safe mode with a non-existing MAC-address zero. The obtained error was reported to the manufacturer who noticed a software bug in the IED and corrected the problem.

6.1. Discussion

During the project a lot of difficulties were faced. The first idea and starting point of the thesis was to program the genetic algorithms in Matlab and convert the code into VHDL by a translator program. Early on it was excluded to use Matlab's own toolbox because of the price of a single licence, so the choice fell on the open source program *Math2Mat* that manages to translate Octave (an open source system capable of executing Matlab coed) into VHDL. However, Math2Mat was also pretty early considered as a bad choice. The use of random numbers and arrays is an essential part of genetic algorithms and in generating random Ethernet frames in a proper way. The software proved to be unable to interpret Octave's *rand* command and was also lacking support for arrays. The final decision was to totally forget the Matlab/Octave conversion and directly program in VHDL or Verilog, even though I had no experience in those languages from the past.

Next major difficulty was the generation of random numbers within the hardware. Random numbers are required for the random initial population and in the genetic algorithms with recombination and mutation, so there had to be an easy way of accessing a random number from a single module. That resulted in the 32-bit LFSR which proved to be an excellent and reliable approach to the problem. Despite the high speed of the FPGA there is seldom a random number repeating when the 32-bit LFSR has $4.3 \cdot 10^9$ cycles before it has looped trough all combinations.

Other major difficulties were the array sizes and the combination with the EtherTester. Since the maximum input/output array size of the FPGA is 32-bit; the generation of the payload, the transmission, the fitness sorting and the GA operations had to be made 32-bit wide. The smoothest and memory efficient way of applying genetic algorithms to the payload was to choose a random 32-bit part and select that part for optimisations. Later on, it was also realised that full 1600 byte payload was not applicable on the FPGA DE4 board, because the GA hardware combined with the EtherTester and with full payload

size extends the total memory on board. To fit both systems on board, a payload size of 160 byte were applied.

When starting the project, the belief of finding harmful data with genetic algorithms was quite low. The biggest concern was how to measure the packages transmission time and how to retrieve the measured times as fitness values. It finally sorted out well thanks to Mika Ruohonen, who helped us to program the Raspberry Pi device for feedback measurements. Although the final results of the GA were quite expected we still managed to create a fast and robust frame transmitter and by other testing also find a major error to a leading manufacturer's IED.

6.2. Future work

Regarding this research, there are a few points that could need improvement and some that could totally be excluded. Some thoughts that have occurred during the research and some different alternatives for further research are:

1. Feedback directly from the FPGA.

   No need for an external measuring device and would improve remarkably in speed. It is however a complex challenge and would require good FPGA programming skills and tremendous knowledge about the IEC 61850 standard and GOOSE protocol.

2. Ignore the GA operation.

   Genetic algorithms seemed to have no impact in finding more harmful data in the payload of the Ethernet frame. However, GA could be extended to also modify other fields in the Ethernet frame.

3. Extended version of the hardware.

   Using an external memory with the FPGA would enable a larger population size and full 1600 byte payload size for each individual.

4. Testing on other IEDs.

   Testing was applied only on two protection relays of the same manufacturer. Extending testing to other IEDs requires reconfiguring of the communication protocol on the Raspberry Pi.

REFERENCES

Alander, Jarmo (2006). *Geneettisten Algoritmien Mahdollisuudet* [pdf]. University of Vaasa.

Alander, Jarmo (2010). Genetic Algorithms, course slides [online]. University of Vaasa. [cited 7.10.2013]. Available on the Internet: <http://lipas.uwasa.fi/~TAU/AUTO3070/slides.php>

Altera (2012a). *Using Triple-Speed Ethernet on DE4 Boards* [online]. Altera Corporation – University Program. [cited 1.10.2013]. Available on the Internet: <ftp://ftp.altera.com/up/pub/Altera_Material/12.0/Tutorials/DE4/using_triple_speed_ethernet.pdf>

Altera (2012b). *DE4 User Manual* [online]. Altera Corporation – University Program. [cited 7.10.2013]. Available on the Internet: <ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Boards/DE4/DE4_User_Manual.pdf>

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

Ciletti, Michael D. (2011). *Advanced Digital Design with the Verilog HDL*. Second edition. Pearson Higher Education. Upper Saddle River, New Jersey. ISBN 978-0-13-601928-2

De Jong, Kenneth A. (2006). *Evolutionary Computation. A Unified Approach*. Massachussetts Institute of Technology. ISBN 0-262-04194-4.

Eiben, A.E. & Smith, J.E. (2007). *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, Germany. ISBN 3-540-40184-9.

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

IEC 61850-1 (2003). *Communication networks and systems in substations – Part 1: Introduction and overview*, 1st edition.

IEC 61850-9-2 (2011). *Communication networks and systems for power utility automation – Part 9-2: Specific communication service mapping (SCSM) – Sampled values over ISO/IEC 8802-3*, 2nd edition.

IEEE (2012). *IEEE Standard for Ethernet* [online]. The Institute of Electrical and Electronics Engineers, inc. New York, USA. [cited 3.10.2013]. Available on the Internet: <http://standards.ieee.org/about/get/802/802.3.html>

Jaakohuhta, Hannu (2005). *Lähiverkot – Ethernet*. 4th edition. Edita Publishing Oy, Helsinki. ISBN 951-826-787-1.

Kriger, Carl, Shaheen Behardien & John-Charly Retonda-Modiya (2013). *A Detailed Analysis of the GOOSE Message Structure in an IEC 61850 Standard-Based Substation Automation System* [cited 28.11.2013]. Available from Internet: <http://univagora.ro/jour/index.php/ijccc/article/view/329/pdf_66>

Mitchell, Melanie (1998). *An Introduction to Genetic Algorithms*. Massachusetts Institute of Technology. London, England.

Spurgeon, Charles E (2000). *Ethernet: The Definitive Guide*. O'Reilly & Associates, Inc. USA. ISBN 1-56592-660-9.

Stevens, Richard W (1994). *TCP/IP Illustrated, Volume 1. The Protocols*. Addison-Wesley Longman, Inc. ISBN 0-201-63346-9.

Söderbacka, Christian (2013). *The GOOSE protocol*. Master's Thesis, University of Vaasa.

TehoFPGA-I (2012). *TehoFPGA-I - Revisoitu hakemus Pohjanmaan liitolle.* Version 2.2. University of Vaasa.

Xilinx (1996). *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators* [online]. [cited 28.10.2013]. Available from Internet: <http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf>

Zeidman, Bob (2006). *All about FPGAs* [cited 3.12.2013]. EETimes University. Available from Internet: <http://www.eetimes.com/document.asp?doc_id=1274496>

Zwolinski, Mark (2004). *Digital System Design with VHDL*. 2nd edition. Pearson Education Limited. ISBN 0-130-39985-X

APPENDICES

APPENDIX 1. System structure

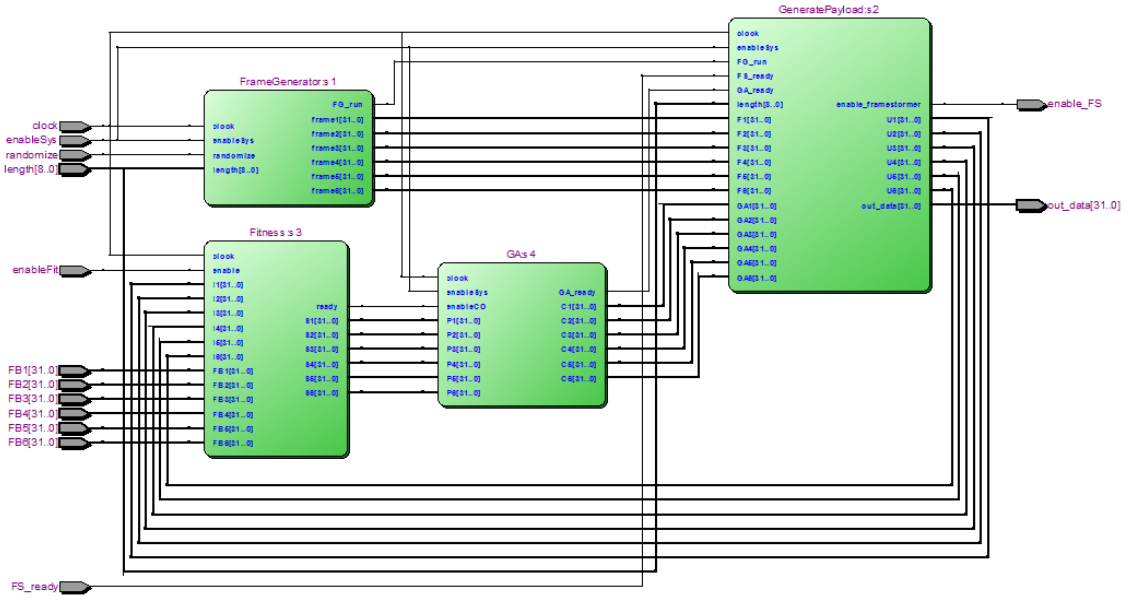


Figure 27. RTL-view of top-level design ***System.v***. The FrameGenerator generates the initial population that is saved to internal registers in the GeneratePayload module. A random part of each frame is selected in GeneratePayload and sent to the Fitness module. The GeneratePayload module transmits the entire population sequentially to *out_data*, and feedbacks from each frame are received to *FB1* to *FB6*. When all feedbacks have been received, the *enableFit* input triggers the Fitness selection, sorts the frames, and passes them over to the GA module. Finally, when GA is done, new frames are replaced in the same position in the GeneratePayload registers (see Figure 15), and the process is repeated again.

APPENDIX 2. GA module

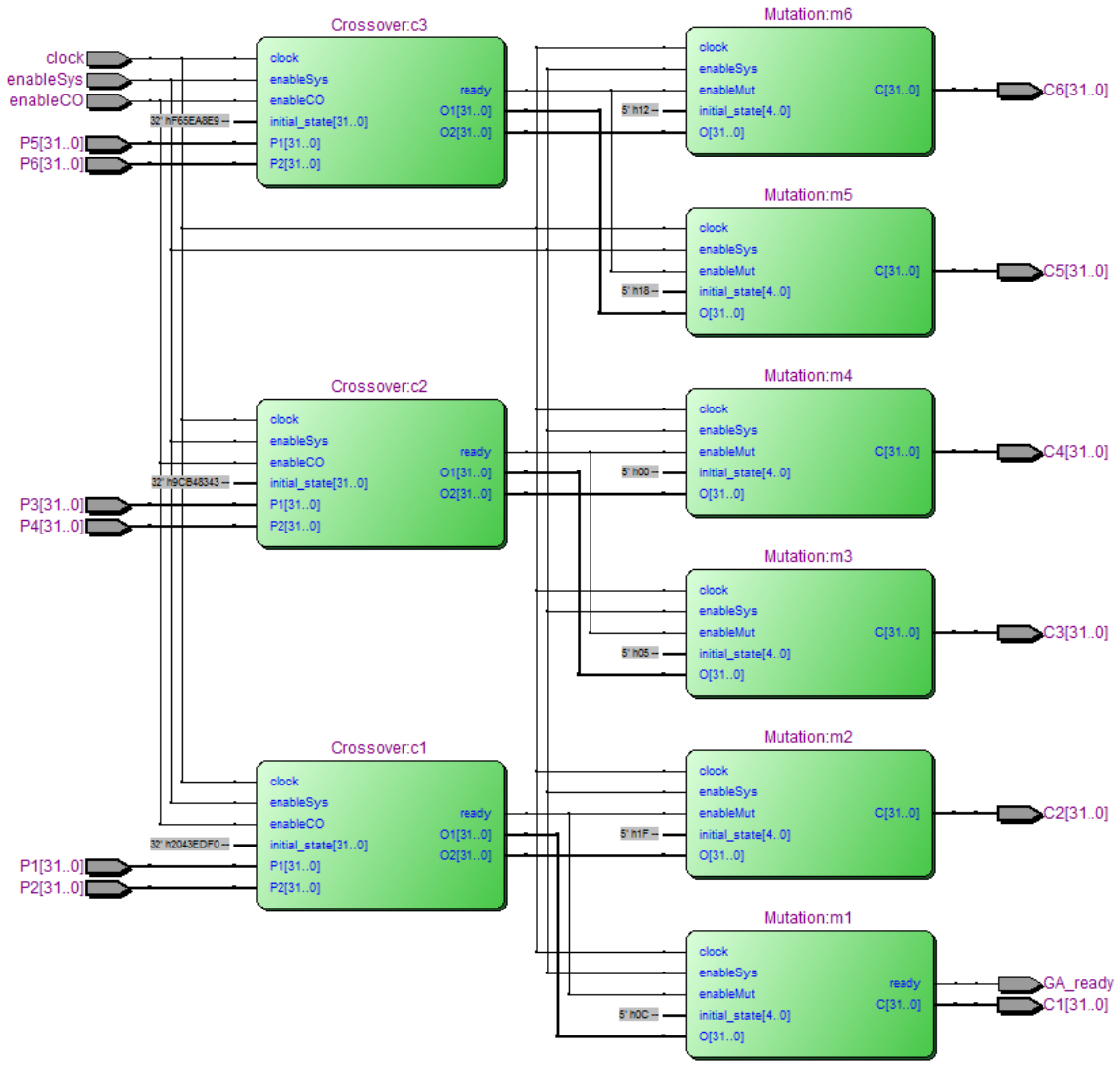


Figure 28. RTL-view of sub-module ***GA.v***. The sorted frames are taken as input-pairs to the crossover modules (*c1 to c3*). In *c1*, parent 1 and parent 2 (*P1* and *P2)* are the frames with the highest fitness values and will be recombined as offspring 1 and offspring 2 (*O1* and *O2)*. Each offspring is processed with separate mutation modules and results in a child. All operations are done simultaneously; therefore the *GA_ready* signal is assigned only from module *m1*. Notice that each module hold different initial states that starts shifting on input *enable_sys*. The *enable_CO* is a ready triggered signal from the previous Fitness selection module.

APPENDIX 3. Verilog code

LFSR.v

```
// - Linear Feedback Shift Register module -
// 32-bit random number generation with LFSR. Starting state of the
// LFSR is determined by the input parameter initial_state. The output
// state of the LFSR is accessed on input randomize.

module LFSR(
    clock,
    enable,
    randomize,
    length,
    initial_state,
    frame,
    FG_run
    );

    parameter N = 32;               // bit length
    input clock;
    input enable;                   // enable LFSR with initial values
    input randomize;                // randomize LFSR
    input [8:0] length;             // payload length
    input [N-1:0] initial_state;    // initial state for LFSR
    output reg [N-1:0] frame;       // output random frame
    output reg FG_run;              // random frame generation running

    reg state;                      // 0 = idle, 1 = run
    reg gen;                        // generate
    reg [8:0] count;
    reg [N-1:0] LFSR;               // LFSRegister

    always @ (posedge clock) begin
        if (enable) begin
            LFSR <= initial_state;
            state <= 1'b1;
            FG_run <= 1'b0;
            count <= 9'b0;
        end
        else if (state) begin
            // XNOR at positions, feedback to MSB.
            // Tap positions at 1,2,22,32.
            LFSR[N-1] <= LFSR[N-1] ^~ LFSR[N-2] ^~ LFSR[N-22] ^~
            LFSR[N-32];
            LFSR[N-2:0] <= LFSR[N-1:1];  // shift to right
            if (randomize) begin
                gen <= 1'b1;
            end
            else if (gen) begin
                if (count < length) begin
                    FG_run <= 1'b1;
                    frame <= LFSR;
                    count <= count + 1'b1;
                end
                else begin
```

```
                    FG_run <= 1'b0;
                    state <= 1'b0;
                    gen <= 1'b0;
                end
            end
        end
    end
endmodule
```

Fitness.v

```
// - Fitness selection module -
// Frame sorting is based on the Bubble-Sort algorithm in descending
// order, according to fitness value. The frame with greatest fitness
// value sorted to output S1, second greatest to S1, etc.

module Fitness(
    clock,
    enable,
    I1,I2,I3,I4,I5,I6,
    FB1,FB2,FB3,FB4,FB5,FB6,
    S1,S2,S3,S4,S5,S6,
    ready
    );

    parameter N = 6;                                // number of frames
    parameter bits = 32;                            // frame size

    input clock;
    input enable;                                   // control by NIOS
    input [bits-1:0] I1, I2, I3, I4, I5, I6;        // input frames
    input [31:0] FB1, FB2, FB3, FB4, FB5, FB6;      // input feedbacks

    output [bits-1:0] S1, S2, S3, S4, S5, S6;       // output frames
    output reg ready;                               // output ready

    reg [31:0] A[1:N];                              // feedback sorting
    reg [bits-1:0] B[1:N];                          // frame sorting
    reg [31:0] i;                                   // loop variable
    reg swap;                                       // 0 = idle, 1 = swap

    assign S1 = B[1], S2 = B[2], S3 = B[3];         // output frames
    assign S4 = B[4], S5 = B[5], S6 = B[6];


    // BUBBLE-SORT
    always @ (posedge clock) begin
        if (enable) begin // read input feedbacks and input frames
            A[1] <= FB1; A[2] <= FB2; A[3] <= FB3;
            A[4] <= FB4; A[5] <= FB5; A[6] <= FB6;
            B[1] <= I1; B[2] <= I2; B[3] <= I3;
            B[4] <= I4; B[5] <= I5; B[6] <= I6;
            i <= 1;     // reset counter
            swap <= 1;  // swap true
        end
```

```
        else if (swap) begin
            if (i < N) begin
                if (A[i] < A[i+1]) begin
                    A[i] <= A[i+1];
                    A[i+1] <= A[i]; // swap feedbacks
                    B[i] <= B[i+1];
                    B[i+1] <= B[i]; // swap frames
                end

                // Check decending order
                else if (A[1] >= A[2] && A[2] >= A[3] && A[3] >= A[4]
&& A[4] >= A[5] && A[5] >= A[6]) begin
                    ready <= 1;
                    swap <= 0;
                end
            end
            i <= i + 1; // increase counter i
        end
        if (i == 6) // stop & repeat at second last position in array
            i <= 1;
        if (ready == 1) // create trigger impulse
            ready <= 0;
    end
endmodule
```

Crossover.v

```
// - Crossover module -
// 32-bit crossover of input parents P1 and P2, according to crossover
// index c_index. Generation of crossover index is based on the same
// method as in the LFSR.v module.
// New offspring are generated to outputs O1 and O2.

module Crossover(
    clock,
    enableSys,
    enableCO,
    initial_state,
    P1,
    P2,
    O1,
    O2,
    ready
    );

    parameter N = 32;

    input clock;
    input enableSys;                    // enable system
    input enableCO;                     // enable crossover
    input [N-1:0] initial_state;        // initial state for LFSR
    input [N-1:0] P1;                   // parent 1
    input [N-1:0] P2;                   // parent 2
    output [N-1:0] O1;                  // offspring 1
    output [N-1:0] O2;                  // offspring 2
    output reg ready;
```

```
    reg state;
    reg [N-1:0] LFSR;                   // internal LFSR
    reg [N-1:0] c_index;

    // CROSSOVER
    assign O1 = (P2 & c_index) ^ (P1 & ~c_index);
    assign O2 = (P1 & c_index) ^ (P2 & ~c_index);

    always @(posedge clock) begin
        if (enableSys) begin
            LFSR <= initial_state;
            state <= 1'b1;
            ready <= 1'b0;
        end
        else if (state) begin
            // XNOR at positions, feedback to MSB.
            // Tap positions at 1,2,22,32.
            LFSR[N-1] <= LFSR[N-1] ^~ LFSR[N-2] ^~ LFSR[N-22] ^~
            LFSR[N-32]; // 32-bit
            LFSR[N-2:0] <= LFSR[N-1:1]; // shift to right
            if (enableCO) begin
                ready <= 1'b1;
                c_index <= LFSR;
            end
        end
        if (ready == 1'b1)
            ready <= 1'b0;  // create ready triggered signal
    end
endmodule
```

Mutation.v

```
// - Mutation module -
// Mutation of input offspring O according to mutation index
// mut_index. Generation of mutation index is based on the same
// method as in the LFSR.v module. New child is generated to output C.
// Mutation rate = 0.03125.

module Mutation(
    clock,
    enableSys,
    enableMut,
    initial_state,
    O,
    C,
    ready
    );

    parameter N = 32;                   // bit length
    parameter R = 5;                    // random number array size

    input clock;
    input enableSys;                    // system enable
    input enableMut;                    // enable mutation module
```

```
input [R-1:0] initial_state;        // initial value
input [N-1:0] O;                    // offspring

output [N-1:0] C;                   // child
output reg ready;                   // GA ready

reg state;                          // LFSR running at high
reg [R-1:0] LFSR;                   // internal LFSR
reg [N-1:0] mut_index;              // mutation index vector

assign C = O ^ mut_index;           // MUTATE

always @ (posedge clock) begin
    if (enableSys) begin
        LFSR <= initial_state;
        state <= 1'b1;
        ready <= 1'b0;
    end
    else if (state) begin
        // XNOR at positions, feedback to MSB
        LFSR[R-1] <= LFSR[R-3] ^~ LFSR[R-5]; // tap pos at 3 and 5
        LFSR[R-2:0] <= LFSR[R-1:1];          // shift to right
        if (enableMut) begin
            ready <= 1'b1;
            case (LFSR)
                // Depending on random number (0-31)
                //the mutation index get a one in that position
                5'h0: mut_index <= 32'h80000000;
                5'h1: mut_index <= 32'h40000000;
                5'h2: mut_index <= 32'h20000000;
                5'h3: mut_index <= 32'h10000000;
                5'h4: mut_index <= 32'h8000000;
                5'h5: mut_index <= 32'h4000000;
                5'h6: mut_index <= 32'h2000000;
                5'h7: mut_index <= 32'h1000000;
                5'h8: mut_index <= 32'h800000;
                5'h9: mut_index <= 32'h400000;
                5'hA: mut_index <= 32'h200000;
                5'hB: mut_index <= 32'h100000;
                5'hC: mut_index <= 32'h80000;
                5'hD: mut_index <= 32'h40000;
                5'hE: mut_index <= 32'h20000;
                5'hF: mut_index <= 32'h10000;
                5'h10: mut_index <= 32'h8000;
                5'h11: mut_index <= 32'h4000;
                5'h12: mut_index <= 32'h2000;
                5'h13: mut_index <= 32'h1000;
                5'h14: mut_index <= 32'h800;
                5'h15: mut_index <= 32'h400;
                5'h16: mut_index <= 32'h200;
                5'h17: mut_index <= 32'h100;
                5'h18: mut_index <= 32'h80;
                5'h19: mut_index <= 32'h40;
                5'h1A: mut_index <= 32'h20;
                5'h1B: mut_index <= 32'h10;
                5'h1C: mut_index <= 32'h8;
                5'h1D: mut_index <= 32'h4;
                5'h1E: mut_index <= 32'h2;
```

```
                    5'h1F: mut_index <= 32'h1;
                endcase
            end
        end
        if (ready == 1'b1)
            ready <= 1'b0; // create ready triggered signal
    end
endmodule
```