NTNU – Trondheim
Norwegian University of
Science and Technology

FPGA Implementation and Evaluation of a Genetic Algorithm for Digital Adaptive Nulling using Space-Time Adaptive Processing.

Mathias By

Master of Science in Electronics
Submission date: June 2015
Supervisor: Snorre Aunet, IET
Co-supervisor: Eirik Kile, Kongsberg Defence Systems

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Summary

By using multiple antennas it is possible to do beam forming of signal reception. Each antenna signal is scaled and delayed before they are added together. By scaling and delaying the signals from various antennas differently, the reception beam form is changed. This scaling and delaying can be done by multiplying each antenna signal with a complex number, this complex number is called a weight in this thesis.

Adaptive nulling means dynamically changing the weights to avoid signal reception from undesired directions based on the received signal. When receiving GPS signals, the desired signals power level is usually lower than the noise floor. If one assumes that there are enough GPS satellites, one can improve the SNR by finding power minimizing weights. This is equivalent to trying to remove all signal reception from directions of powerful signals. The optimal Wiener solution can be found by the direct matrix inversion, DMI, method. The DMI method is complex and large parts of the method can not be done in parallel. The goal of this thesis is to determine whether an FPGA implementation of the genetic algorithm, an iterative random search algorithm, realistically can replace the DMI method. The target FPGA in this thesis is Xilinx Virtex 6 XC6VLX195T.

A MATLAB model of the genetic algorithm is developed. This model together with some assumptions is used to find reasonable parameters for the genetic algorithm. The resulting algorithm is implemented in VHDL. The VHDL implementation is partially tested, synthesised and run through place and route. The genetic algorithm module is supposed to be part of a bigger FPGA design. The amount of inputs and outputs in the genetic algorithm module makes it impossible to route the design by itself on the target FPGA. The design is wrapped to solve this problem. The maximum clock frequency of the wrapped design is 180 MHz after place and route.

It turns out that the weights found by the genetic algorithm is far from the optimal Wiener solution, which is the theoretically best weights that can be found. It does not seem likely that the genetic algorithm can compete with the DMI algorithm in this real time scenario. The achieved performance of the genetic algorithm is a received power reduction of around 20 dB. When using the DMI method the power reduction is 50-60 dB. Although it has been shown that the performance tests of both the genetic algorithm and the DMI method were not ideal, there is a major performance difference. The genetic algorithm finds new weights faster than the DMI method. However the gap between the two methods in quality of the weights is presumed too large to be closed.

Sammendrag

Ved å bruke mer enn en antenne er det mulig å påvirke loben til signalmottaket. Signalene fra antenne summeres, men forsinkes og skaleres ulikt for å oppnå et ikke-uniformt signalmottak. Det å forsinke og skalere vektene gjøres i praksis ved å multiplisere det mottatte signalet fra hver antenne med et komplekst tall. De komplekse tallene brukt til å skalere og forsinke antennesignalene kalles vekter i denne rapporten.

Det er mulig å unngå å motta signal fra retninger som er uønsket ved å endre på vektene. Hvis man baserer hvilke retninger som er uønsket på det mottatte signalet kalles dette adaptiv nulling. Når man tar imot GPS-signaler har GPS-signalet som regel mindre effekt enn støygulvet. Det er mulig å forbedre mottaksforholdene for GPS signalet ved å unngå å motta signal som har effekt over støygulvet. Gitt at det er nok GPS satellitter i andre retninger, lønner det seg å prøve å nulle signal mottaket fra de retningene hvor det befinner seg kilder som sender mye effekt. Dette tilsvarer å finne vekter som gjør total mottatt effekt så liten som mulig. Det er mulig å regne seg fram til den løsningen som minimerer mottatt effekt, den optimale Wiener løsningen, med en metode kalt direkte matrise invertering, eller DMI. DMI metoden krever matrise invertering som er en kompleks og regnekrevende operasjon. Det er også en stor del av DMI metoden som ikke kan gjøres i parallell. Må let med denne oppgaven er å se om en FPGA implementasjon av en genetisk algoritme realistisk sett kan erstatte DMI algoritmen. FPGAen som er tenkt brukt for den endelige kretsen er en Xilinx Virtex 6 XC6VLX195T.

For å måle ytelsen til den genetiske algoritmen er det utviklet en MATLAB modell. Denne modellen og sunn fornuft er brukt for å finne en fornuftige parametere for den genetiske algoritmen. Den resulterende algoritmen er implementert i VHDL. De aller fleste modulene i denne kretsen er testet og verifisert. VHDL designet er også syntetisert, komponentene som skal brukes i FPGAen er valgt og designet er rutet. VHDL designet i denne oppgaven er ment å være en del av et større VHDL design. Antall innganger og utganger i designet gjør at det ikke passer med den FPGAen som skal brukes når den skal plasseres alene. Derfor er det lagt annen logikk rundt oppgavens design slik at det ikke lenger har for mange innganger og utganger for å passe i FPGAen. Oppnådd maks klokke rate med designet som er pakket inn i annen logikk er 180 MHz ferdig plassert og rutet.

Det viser seg at spranget mellom hvor gode vekter den genetiske algoritmen klarer å finne og de optimale vektene fra DMI algoritmen er veldig stort. Den genetiske algoritmen klarer å redusere signalstyrken i test datasettet med 20 dB. Til sammenligning reduseres effekten med 50-60 dB for DMI løsningen. Det er vist at ytelsestestene ikke har vært presise nok. Men på grunnlag av at avstanden i kvalitet på vektene er så stor virker det ikke sannsynlig at den genetiske algoritmen kan erstatte DMI algoritmen.

Preface

This master thesis was completed at the Department of Electronics and Telecommunications at the Norwegian University of Science and Technology during the spring of 2015. It was written as the final part of the master program at NTNU.

I would like to thank my supervisor Snorre Aune for advice during this thesis. But I would specially like to thank Eirik Kile, Erik Narverud and Håvard Sannes from Kongsberg Defence & Aerospace for providing the problem and for valuable advice throughout the design process.

Table of Contents

List of Tables

List of Figures

Abbreviations

FPGA	=	Field Programmable Gate Array
VHDL	=	Very high speed integrated circuit Hardware Description Language
DSP	=	Digital Signal Processor
SNR	=	Signal-to-Noise Ratio
MUX	=	MUltipleXer
ADC	=	Analog-to-Digital Converter
DMI	=	Direct Matrix Inversion
GPS	=	Global Positioning System

Chapter 1

Introduction and Motivation

This chapter is an introduction to the problem that is to be solved in this thesis. The structure of the project and the structure of the report is also described.

1.1 Space-Time-Adaptive Processing of GPS Signals

When receiving radio signals, one might experience that undesired signals arrive from certain directions. To improve the SNR it is then desirable to reduce signal reception from these directions while maintaining signal reception from all other direction. This can be made possible by having an array of antennas [4]. From these antennas, signals with different delays, called taps, are used. See figure 1.1. By scaling and phase shifting these taps with weights before adding them together, one can use destructive interference to reduce reception of signals from the undesired directions.

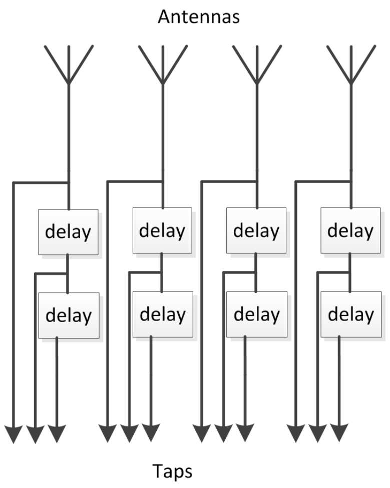


Figure 1.1: Antenna system.

When receiving some types of radio signals the information signal have less power than the noise floor. This is normally the case for GPS signal reception. To improve the SNR when receiving GPS signals, one can try to reduce signal reception in the directions where the most power is received. This means finding weights for the taps that minimize the total received power. By reducing signal reception in the directions where a lot of power is received, one also reduces the reception of GPS signals from these directions. The GPS signals from the satellites that are located in the received directions will get a better SNR. A better SNR will simplify the acquisition of the GPS signals. If enough GPS satellites are located in the received directions, this will improve the GPS reception. One of the antennas, often called the reference antenna, have a constant weight. By forcing one tap to be received one ensures that the solution; all weights zero, is not possible. The process of reducing signal reception in certain directions based on the received signal, using an antenna array and tapped delay lines, is called space-time-adaptive nulling[4].

1.2 Defining Signals and Correlation Matrices

Let $x_a(t)$ be the signal output from the a'th out of n non-reference taps at time t. These signals are placed in the received signal vector $\mathbf{x}(t)$ as seen in (1.1).

$$\mathbf{x}(t) \triangleq \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix} \tag{1.1}$$

Further, the auto-correlation matrix for time t, $\mathbf{r}_{xx}(t)$ is shown in (1.2) [4].

$$\mathbf{r}_{xx}(t) = E[\mathbf{x}(t) \cdot \mathbf{x}^{\dagger}(t)] \tag{1.2}$$

In (1.2) † is the hermitian conjugate and $E[v]$ is the expected value of v. Let $d(t)$ be the received signal from the reference antenna at time t. The cross-correlation between the reference signal and the signal from the other taps at time t, $\mathbf{r}_{xd}(t)$ is defined in (1.3) [4].

$$\mathbf{r}_{xd}(t) = \begin{bmatrix} \overline{x_1(t) \cdot d(t)} \\ \overline{x_2(t) \cdot d(t)} \\ \vdots \\ \overline{x_n(t) \cdot d(t)} \end{bmatrix} \tag{1.3}$$

In (1.3) $\overline{x_a(t) \cdot d(t)}$ denotes $E[x_a(t) \cdot d(t)]$. The cross-correlation matrix and the auto-correlation matrix can be used to find the power minimizing weights. This is shown in section 1.3.

1.3 Direct Matrix Inversion

One algorithm used to find the weights to minimize the power of the weighted result is direct matrix inversion, DMI. $w_a(t)$ denotes the weight for signal $x_a(t)$. The weights are put in a vector $\mathbf{w}(t)$ as seen in (1.4).

$$\mathbf{w}(t) \triangleq \begin{bmatrix} w_1(t) \\ w_2(t) \\ \vdots \\ w_n(t) \end{bmatrix} \tag{1.4}$$

$\mathbf{w}_{opt}(t)$ denotes the power minimizing weights. $\mathbf{w}_{opt}(t)$ can be found using the Wiener-Hopf equation in matrix form, shown in (1.5) [4].

$$\mathbf{r}_{xx}(t) \cdot \mathbf{w}_{opt}(t) = \mathbf{r}_{xd}(t) \tag{1.5}$$

$\mathbf{w}_{opt}(t)$ is often called the optimum Wiener solution. $\mathbf{r}_{xx}(t)$ is the auto-correlation matrix and $\mathbf{r}_{xd}(t)$ is the cross-correlation matrix. By rearranging (1.5) one gets (1.6).

$$\mathbf{w}_{opt}(t) = \mathbf{r}_{xx}^{-1}(t) \cdot \mathbf{r}_{xd}(t) \tag{1.6}$$

To find the optimum Wiener solution by directly calculating this expression of (1.6), which involves matrix inversion, is called the direct matrix inversion method. In this thesis it is also referred to as the analytic solution.

1.4 Problem Description

Matrix inversion is a complex process and a lot of the calculations can not be done in parallel [6]. This makes it slow even if one use a hardware accelerator. The goal of this thesis is to see if it can be beneficial to use an FPGA to find weights using an iterative algorithm. The algorithm used should be more easily pipelined and parallelized compared to the DMI. The performance should be compared with the performance of the DMI algorithm.

1.5 Proposed algorithm

The proposed algorithm is a genetic algorithm, which is an iterative search algorithm that mimics the process of evolution [3]. The typical problem solved with a genetic algorithm has an enormous solution space; the genetic algorithm is used to find a good solution by gradually exploring the solution space. The search is usually focused around the so far best solution. Different versions of the genetic algorithm have different spread in this search around the so far best solution.

1.6 Existing Solutions

FPGA implementations of the genetic algorithm have been done before [1, 7]. In this thesis the genetic algorithm is used to solve a complex problem. The goal of this thesis is to see if an FPGA implementation of the genetic algorithm is capable of solving a complex problem in a real time system. To achieve this hardware solutions that pushes the processing speed of the genetic algorithm is found.

1.7 System specifications

To evaluate if the genetic algorithm can be used instead of the DMI algorithm, a specific scenario is given. A Xilinx Virtex 6 XC6VLX195T FPGA should be used to find weights for a sample input. The number of non-reference antennas in the sample input is three. The number of taps per antenna is three. The average power for the reference antenna, the cross-correlation matrix, the auto-correlation matrix and the generation of the random bits needed in the genetic algorithm are considered outside the scope of this thesis. These signals are assumed available as inputs to the genetic algorithm design. It is preferable for the hardware design to have a maximum frequency of at least 200 MHz.

1.8 Methodology

The nature of the problem, the DMI solution and the suggestion to use a genetic algorithm on an FPGA was part of the project proposal. The problem of this thesis is limited to developing and analysing an implementation of a genetic algorithm. The algorithm should be able to find weights which reduces the power of the received signal. The power reduction of the genetic algorithm should be compared to the power reduction achieved by the ideal weights found by the DMI method for the given sample data. This problem will be solved in several steps. A MATLAB model that is as close as possible to the hardware will be developed. This model is then used together with the sample data set from the GPS antennas to find reasonable parameters for the algorithm. The resulting algorithm is then implemented on an FPGA and verified. The MATLAB model should also be used to measure the performance of the genetic algorithm. The performance should then be compared to the performance of the ideal weights found by the DMI algorithm. The final implementation stage would be to run hardware tests in a real antenna system. An overview of the stages of the project can be seen in figure 1.2.

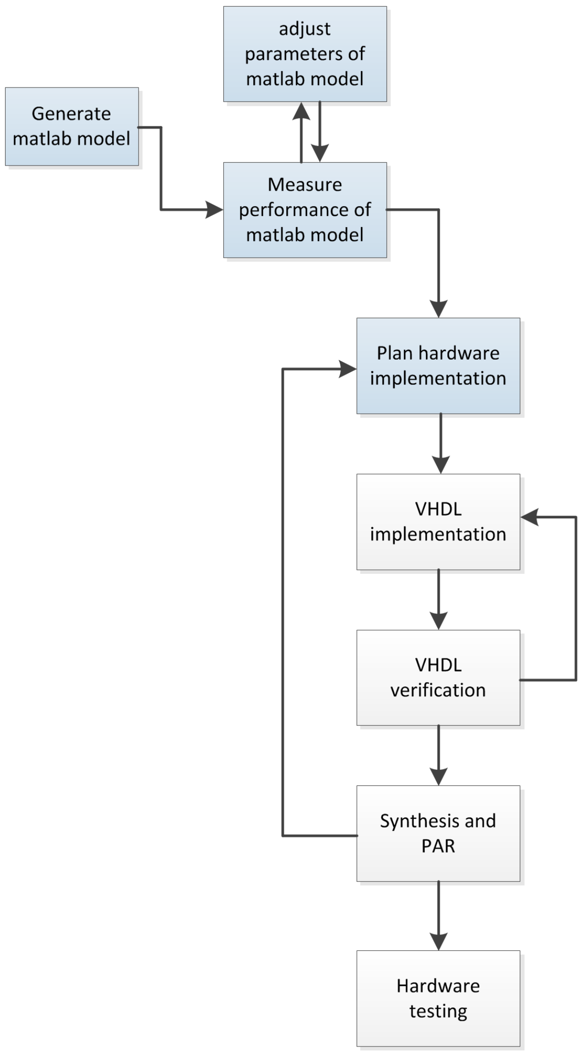

adjust parameters of matlab model
Generate matlab model
Measure performance of matlab model
Plan hardware implementation
VHDL implementation
VHDL verification
Synthesis and PAR
Hardware testing

Figure 1.2: Project overview, the blue boxes are processes that were started during the fall semester 2014.

1.9 Present Contributions

This project started during the fall semester of 2014. The goal of the work done during the fall was to find reasonable parameters and prepare for the hardware implementation of the algorithm. This work was not completed during the fall semester of 2014. The work of the current semester, spring 2015 therefore started with completing the work of finding reasonable parameters for the genetic algorithm.

1.10 Structure of the Thesis

This thesis is divided into seven chapters, a description of the different chapters follows.

- Chapter 2 describes some of the important results of the work done during the fall semester of 2014. The chapter also includes some of the background theory needed for the project.
- Chapter 3 completes the work of finding reasonable parameters for the genetic algorithm. This chapter contains both results and discussion regarding the algorithm that is implemented in this thesis.
- Chapter 4 is used to describe the hardware implementation of the genetic algorithm. The chapter discusses different solutions and it is described what is implemented. The chapter also contains information about the verification and synthesis of the VHDL design. Both discussion and results regarding the hardware implementation of the algorithm is found in this chapter.
- Chapter 5 is the general discussion of this thesis. The performance of the system and the important choices made that can have effected the performance of the genetic algorithm is discussed.
- Chapter 6 is the conclusion of this thesis.
- Chapter 7 discusses future work of the project.

Chapter 2

Background

2.1 Genetic Algorithm

As mentioned in the introduction, the genetic algorithm is an iterative search algorithm that mimics the process of evolution [3]. The algorithm searches the solution space by attempting several possible solutions. One solution is called an individual, the smallest part of a solution is called a gene and a group of solutions is called a population. Populations are changed for each iteration of the algorithm and the population of one iteration is called a generation. There exist several different versions of the genetic algorithm, yet the basis of most versions is one or more populations and a cost function used to evaluate the individuals. The cost function is used to measure how good a solution is and is the basis for the evolutionary processes like breeding, mutation and migration. These processes will be described in section 2.1.1 through 2.1.3. Figure 2.1 shows the flow chart of a typical genetic algorithm.

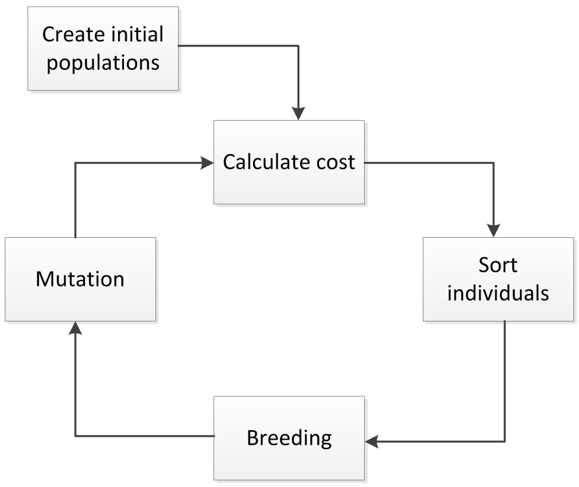


Figure 2.1: Flow chart for genetic algorithm.

2.1.1 Breeding

For each generation the cost function is used to determine which are the best individuals [3]. Some portion, typically the best half, of the population is then paired up. Each pair of individuals are then used to generate new individuals. The new individuals are called children. The pair used to generate children are called parents. One or two children are produced from each pair of parents. The children replace some of the high cost individuals. The genes of the children are randomly picked from the parents. If two offspring are generated, for each gene the two children usually get their genes from one parent each. A good breeding process ensures survival of the low cost individuals and cross the genes of these low cost individuals to create even lower cost children.

2.1.2 Mutation

Mutation is a technique used to avoid that the search gets stuck at a local extreme [3]. The genes of individuals are randomly mutated to make sure that the search covers the whole solution space. The best individual of a population is usually not mutated.

2.1.3 Multiple populations and migration

When multiple populations are used, the different populations usually serve different purposes [3]. For instance, one population can explore the whole solution space randomly. The best individuals from this population might then migrate to another population. This other population can then search a smaller area around the migrated solution. Different schemes for both the purpose of the different populations and the migration have been researched.

2.2 Structure of the Xilinx Virtex 6 XC6VLX195T FPGA

The target FPGA for this thesis, the Xilinx XC6VLX195T has 31200 slices and 640 DSP48E1 [8].

2.2.1 FPGA Slice

A slice is what Xilinx call one fraction of the FPGAs most configurable logic [8]. One slice consists of four look up tables (LUTs), 8 registers, MUXes and arithmetic carry logic. Since Xilinx XC6VLX195T has 31200 slices it has 249600 slice registers and 124800 slice LUTs.

LUT

A look up table or a LUT can be used to solve any logic function within its limit of inputs and outputs [8]. In the Virtex 6 series, Xilinx use LUTs with six inputs and one output. With common inputs, one six input LUT can be used as two five input LUTs with one output each.

DSP48E1

The DSP48E1 slices, commonly called DSPs, are customisable digital signal processing blocks [8]. The DSPs consists of a dedicated 25 x 18 bit two's compliment multiplier and a 48-bit accumulator. The DSPs can be configured to do all sorts of tasks and are commonly used for multiplication, subtraction, and division.

2.3 Hardware Implementation of Genetic Algorithm

Since the purpose of this study is to find a fitting version of the genetic algorithm for an FPGA, it is important to develop the algorithm with that in mind. This means setting the parameters for the algorithm such that the algorithm can be implemented on the FPGA.

2.3.1 Hardware Implementation of Mutation

For the genetic algorithm to work it is important that random bits are mutated. Random bits are available as an input to this system. To mutate a bit, a XOR operation between the old bit and an AND operation of a certain amount of random bits can be used. This way of mutating bits can be seen in figure 2.2.

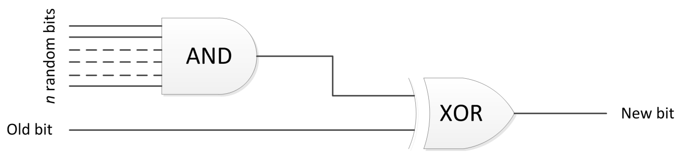


Figure 2.2: Single bit mutation.

The probability of mutation of one bit is $\frac{1}{2^n}$, where n is the number of bits going into the AND function. The number of bits that are used in this logic function is $n + 1$. This number of inputs will determine how many LUTs are needed in the FPGA to mutate one bit. Since a regular LUT has six inputs in the Virtex 6 FPGA, a mutation rate of $\frac{1}{32}$ is the lowest that fits into one LUT. A mutation rate of $\frac{1}{64}$ or less will then use two LUTs. It is therefore beneficial to use mutation rates that is higher than or equal to $\frac{1}{32}$.

2.3.2 Hardware Implementation of Breeding

The parents are paired up two and two, from the best individuals down to the last individuals that can be breed without increasing the population size. Then both the imaginary and the real part of the parents are crossed.

Breeding of genes or crossing two n bit binary numbers can be done by a bitwise MUX operation. A random number is used as the select input and one bit from each parent as the two regular MUX inputs. There exists several MUXes in each cell of the Virtex 6 FPGA.

Some different versions of the breeding process in the genetic algorithm exists, however the major difference is whether the two parents create one or two children.

2.3.3 Hardware Implementation of Cost Function and Sorting

A basis for all the evolutionary processes are a method of ranking the individuals. To rank the individuals the cost needs to be found and used to sort the individuals.

The sample rate of a normal GPS antenna system ADC is around 100 MHz. As will be shown later in this thesis the generation frequency of the genetic algorithm will be a lot lower than this. The input to the genetic algorithm is usually not changed more often then once per generation. A new set of weights are found once per generation. The generation frequency is a lot lower than the ADC sample rate. This means that each set of weights must be used for multiple ADC samples. It is beneficial to find weights that minimize the average power of all the samples that the weights will be used for. It is therefore natural to use contributions from all the ADC samples that the weight will be used for in the cost function. The first method of calculating the cost function is shown in 2.1.

$$cost = \frac{1}{K} \cdot \sum_{k=1}^{K-1} (d(n+k-1) + \mathbf{w}(n+k-1) \cdot \mathbf{x}(n+k-1))^{1+\dagger} \tag{2.1}$$

In (2.1) K is the number of samples that is used in the cost function. $d(t)$ is the signal from the reference antenna at sample t. $\mathbf{w}(t)$ is the weights for sample t. $\mathbf{x}(t)$ is the signals from the non reference antennas at sample t. The $\dagger$ is the Hermitian conjugate, or since this is the Hermitian conjugate of a single complex number it becomes the complex conjugate. In (2.1) $a^{1+\dagger}$ is used as a representation for $a^1 \cdot a^\dagger$ for readability. $d(t) + \mathbf{w}(t) \cdot \mathbf{x}(t)$ is the weighted sum of the signals from the antennas at sample t. This is then multiplied with itself complex conjugated. To multiply a complex signal with itself complex conjugated give the power of the signal [2]. The sum over K samples divided by K is the average. So the cost function is the average power of the weighted signals from the antennas.

Chapter 3

Algorithm for implementation

The genetic algorithm is not a very specific algorithm. It is the idea of mimicking the process of evolution to solve complex problems. The ideal parameters for the genetic algorithm like number and size of populations, mutation rates and migration rates are problem specific. It has not been found well documented guidelines on how to chose parameters for the genetic algorithm in the literature.

The genetic algorithm is an iterative random search algorithm. This means that the results when simulating the algorithm will differ every run. To determine the effect of changing one parameter therefore require multiple simulations. The MATLAB model which is designed to be as close to hardware as possible have a lot of bit operations. There are a large amount of data in the simulations. This data is processed with complex processes and calculations. This makes the simulations very time consuming. The combination of time consuming simulations, variation in results and lack of good guidelines for setting the parameters makes this a typical engineering problem. To find the ideal parameters even just for the sample data set, would take a wast amount of time. The goal of this chapter is not to find ideal parameters for the genetic algorithm. The goal is to find reasonable parameters and to compare the performance of this version of the genetic algorithm with the DMI algorithm performance. When the methods are compared one must then keep in mind that the genetic algorithm used is not the absolute optimal genetic algorithm for this problem.

3.1 Precision

The precision of the weights are important because they determine the width of the arithmetic operations. The precision also sets the limit of how good solutions that can be found by the genetic algorithm. In the prestudy 32 bit IEEE 754 floating point was used. The advantage of floating-point is the dynamic range and that the problems of overflow and underflow is simplified. To find the range of the weights the DMI method were used on the sample data set. By running the DMI method on the sample data set, it was seen that the range of the optimal weights were $< -1, 1 >$. This means that the dynamic range of floating point was not exploited. It was therefore suggested to use signed fixed-point representation with range $< -1, 1 >$ instead. This will both simplify arithmetic operations in general, because fixed-point is a less complex number representation than floating-point, and reduce the number of bits needed for a given precision. To determine how many bits wide fixed-point number were needed, the DMI solution was converted to signed fixed-point with range $< -1, 1 >$. The optimal weights were then truncated to different number of bits. The differently truncated solutions were then compared to the MATLAB double precision floating-point solution. The comparison of the solutions was done using the cost function of the genetic algorithm, which will be defined in section 3.2. A plot of the cost of some differently truncated solutions, as well as the double precision solution, is shown in figure 3.1.

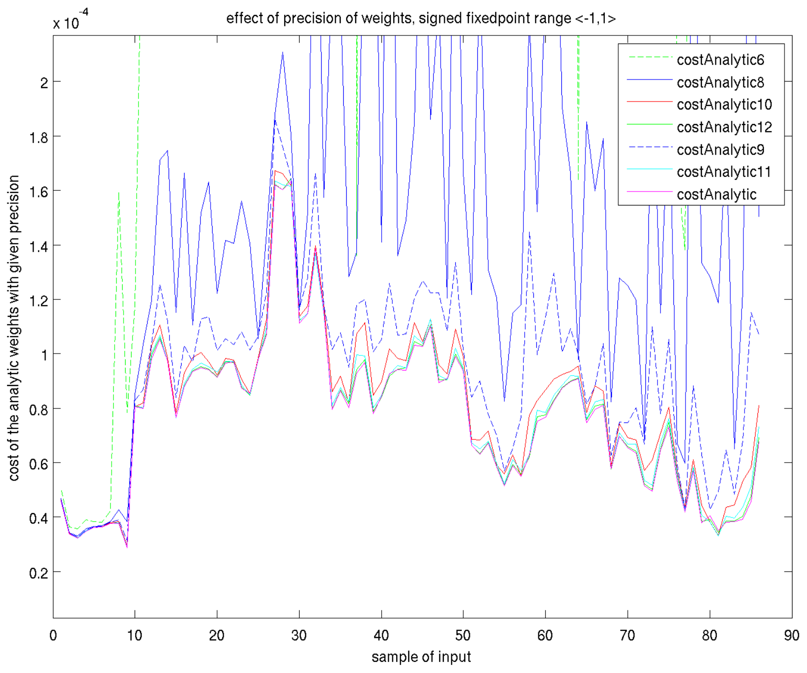


Figure 3.1: Cost of differently truncated optimal weights.

In figure 3.1 costAnalytic is the DMI solution with double precision. costAnalytic$n$ is the DMI solution, where both the imaginary and the real part of the weight is truncated to n bits signed fixed point. The scope of figure 3.1 is set such that the truncated solutions that are close to floating-point solutions can be seen. The solutions that have values outside this scope is considered to have too few bits. One can see from figure 3.1 that the trend is that less bits in the solution leads to a higher cost. This is natural because less bits to represent the optimal solution means a less optimal solution, which has a higher cost. The

DMI solution truncated to ten bits is the most truncated solution that has cost only a few percent above the double precision solution. It is assumed that ten bits signed fixed-point representation is sufficient. If the cost of simulations should become close to the cost of the ten bit truncated DMI solution, adding more bits to the number representation should be considered.

3.2 Cost Function

The cost function for this specific problem is, as described in chapter 1, the power of the weighted sum of the received signals. The first method of calculating the cost function was given in (2.1). This method calculates the average power of the weighted sum of K samples. This is done because the generation rate of the algorithm, most likely, will be a lot lower than the sample rate of the input signals. The weights found by the genetic algorithm will have to be used for more than one data sample. This minimum number of data samples that one set of weights have to be used for is called s. It is natural to use a K that is $l \cdot s$, where l is a positive integer. This will mean that l generations of the genetic algorithm is run, trying to find weights that fit these $l \cdot s$ samples. The weights are then used for all these samples. The K will effect the behavior of the algorithm. It might be beneficial to change K dynamically while the algorithm is running. When calculating the cost function as given in (2.1), the number of multiplications and additions in the cost function calculation is dependant on K. This will make the generation period dependant on K, which can be a disadvantage.

The auto-correlation, $\mathbf{r}_{xx}$, the average power of the K last samples of the reference channel, p_1, and the cross-correlation, $\mathbf{r}_{xd}$, is outside the scope of this thesis and considered available as inputs to the genetic algorithm design. These are based on K samples. For the same K (2.1) and (3.1) will result in exactly the same cost. (3.1) gives the relation between the expected power for the weighted sum of signals and the correlations [5].

$$E[|\ d(t) - \mathbf{w}^\dagger \cdot \mathbf{x}\ |^2] = p_1 - \mathbf{r}_{xx}^\dagger \cdot \mathbf{w} - \mathbf{w}^\dagger \cdot \mathbf{r}_{xx} + \mathbf{w} \cdot \mathbf{r}_{xd} \cdot \mathbf{w} \tag{3.1}$$

The calculation of the expected power using (3.1) does not involve K when $\mathbf{r}_{xx}$, $\mathbf{r}_{xd}$ and p_1 are pre-calculated. This means that changing K will not effect the generation frequency, which is a big advantage if one wants to dynamically change K. The number of arithmetic operations in the cost function is an important factor to which generation frequency that can be achieved. The two different methods of calculating the cost is therefore compared in terms of arithmetic operations per individual. More precisely, the number of complex additions and complex multiplications needed for the calculation of the cost of one individual. The number of these operations are dependant on the number of non-reference taps, AT, and K. The number of complex multiplications in the correlation approach is none for the first term, AT for each of the second and third term and $AT^2 + AT$ for the last term. Number of complex additions is none for the first term, $AT - 1$ for each of the second and last term, $(AT-1)^2 + AT - 1$ for the last term and 3 additions or subtractions to add the terms together. For the sum based cost function found in (2.1), the number of complex multiplications needed for the calculation of one individual are $K \cdot AT$ for the vector product and K for the last multiplicatin inside the sum. The number of additions for the sum based method is $K \cdot (AT - 1)$ for the vector product and K additions for the addition inside the sum. The number of additions in the correlation based method is compared with the number of additions in the sum based method with different values for K in figure 3.2. In the figure k is used for K in the legend.

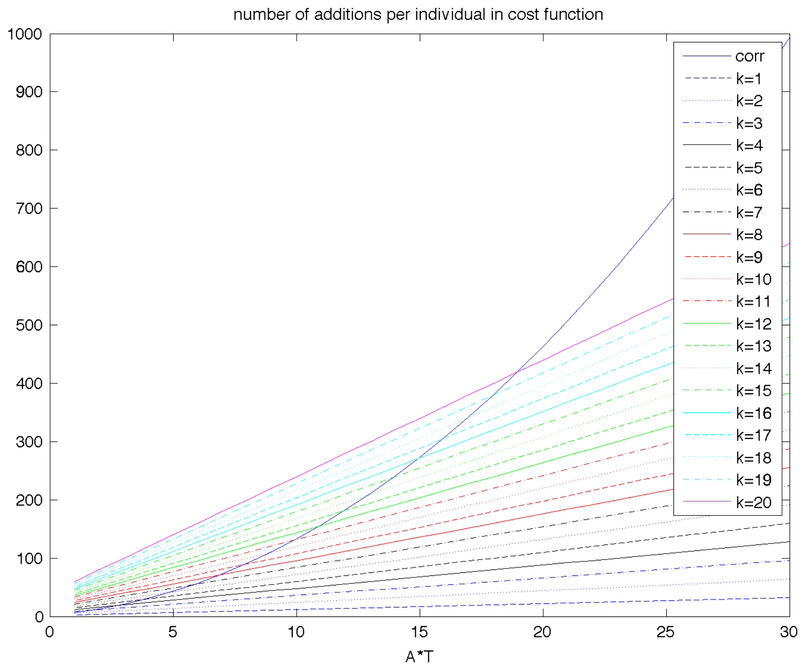


Figure 3.2: Number of complex additions in different methods of calculating the cost function.

As one can see from figure 3.2 the number of additions is lower for the correlation method for a low number of taps and a high number of K. The number of complex multiplications needed for each individual in the different cost calculation methods are shown in figure 3.3.

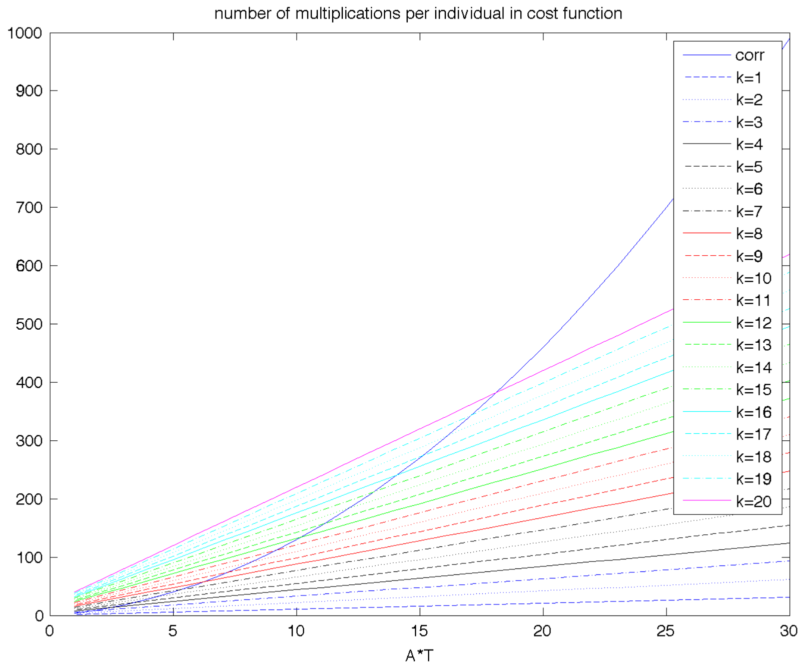


Figure 3.3: Number of complex multiplications in different methods of calculating the cost function.

For the number of multiplications it is the same trend as for the additions. In the sample scenario of this problem of this thesis, the number of non reference taps are nine. The number of multiplications are the most important, because multiplication takes more logic than addition. As one can see from figure 3.3 the number of multiplications when AT is nine is lower for the correlation method, if K is larger than or equal to approximately eleven. It is very likely that the generation period will exceed eleven times the sample rate. The cost functions calculate exactly the same under the assumption of an equal K. When the number of arithmetic operations are estimated for the correlation based cost function calculation method, the calculations needed for p_1, $\mathbf{r}_{xx}$ and $\mathbf{r}_{xd}$ are not included. These calculations are though just needed one time for each time the set of samples used in the cost function calculation is changed. It is not likely to change the samples used in the cost function with a higher rate than the generation frequency. This means that the time available for this calculations is long and that the calculations can be done with a small amount of hardware.

3.3 Mutation rate

Mutation rate can be an input to the genetic algorithm design. The mutation rate is limited to be from $\frac{1}{2^3}$ to $\frac{1}{2^7}$. The mutation rate must be limited to not demand infinite hardware. These specific limits are set because mutation rates outside these limits are considered unreasonable. If the mutation rate is lower than $\frac{1}{2^7}$, in average, less than one bit is mutated per individual. If the mutation rate is higher than $\frac{1}{2^3}$, in average, at least every forth bit is mutated.

3.4 Number of individuals per population

The number migrants per populations, the number of individuals per population and the number of populations effect how well the algorithm works. These three parameters can be constants in the FPGA design that could easily be adjusted before synthesis. However, if the populations sizes are the same size, this will reduce the complexity of the genetic algorithm design. The complexity is reduced because the breeding and selection process can be specialized for a fixed population size. It is therefore performed simulations with the same amount of individuals in four populations. This simulation is compared with simulations where the individuals are differently distributed in the four populations. Populations with mutation rates $\frac{1}{2^4}$, $\frac{1}{2^5}$, $\frac{1}{2^6}$ and $\frac{1}{2^7}$ is used. These populations is intended to serve different purposes. The lowest mutation rate population is the population that explores solutions similar to the best known individual. The other populations, which have higher mutation rates, explores solutions that are more different to the currently best solution.

The simulations have been performed with 100 individuals. K, the number of samples averaged over in the cost function is 20. Two individuals can migrate from every population to every other populations once every generation. The generation period is the same as the sample rate. The sample data set have been run 30 times and the average cost of the 30 runs are plotted. The number of individuals in the lowest mutation rate population is varied. The remaining individuals is evenly distributed in the other populations. Figure 3.4 shows a simulation with 85 individuals in the lowest mutation rate population and five individuals in the other populations.

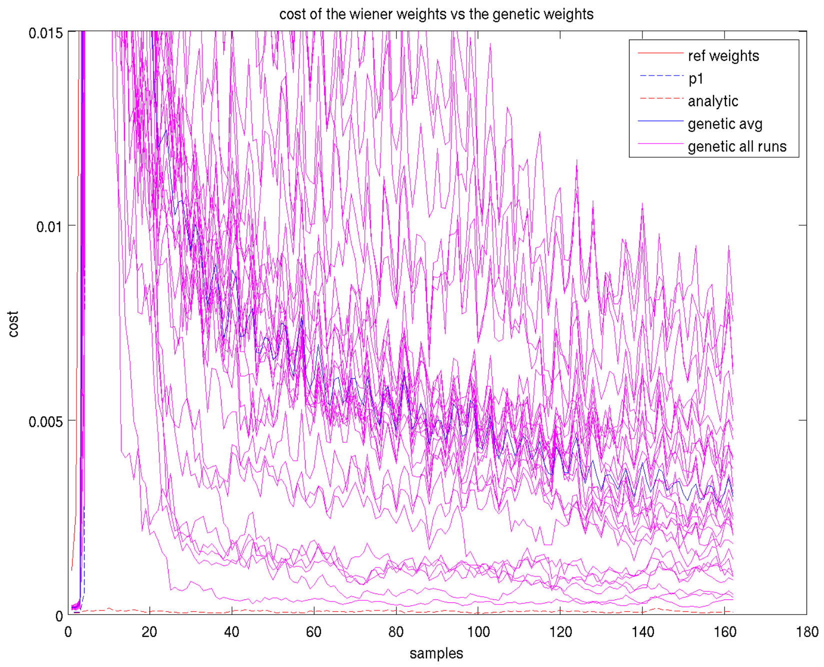


Figure 3.4: Simulation of genetic algorithm with 85 individuals in the lowest mutation rate population.

In figure 3.4 as well as for all the figures of section 3.4 ”p1” is the average power of the last 20 samples of the reference channel. ”analytic” is the cost of the optimal weights found by the DMI method. ”genetic avg” is the average cost of the 30 runs of the sample data set. ”genetic all runs” is all the 30 runs of the sample data set. ”ref weights” are a single population with the same initial conditions as the genetic algorithm popultaion, but no evolutionary processes are used. The best individual of this population is simply chosen for every sample. A jammer seems to be turned on around sample five in the sample set used for the plots. One can see in all the figures of section 3.4, that the cost of the genetic algorithm weights increases rapidly when the jammer is assumed to be turned on. Further it decreases as better weights are found. As one can see in all figures in section 3.4 almost every signal in the figures is at some point outside the scope of the figures. This is done to be able to see how the genetic algorithm performs. Both ”p1” and ”ref weights” can only be seen in the beginning of the simulations, after sample five they are both around 20 dB above the genetic solution. It can be seen that ”p1” is about 20 dB above the genetic solution is figure 3.9. The important matter here is to compare the different versions of the genetic algorithm, with each other, after they have had some time to find reasonable weights after the sudden change.

Figure 3.4 shows the simulation where the most individuals are in the lowest mutation rate population. The average cost of the genetic weights is around 0.004 for the last 20 samples. Figure 3.5 shows the simulation of the genetic algorithm with 76 individuals in the lowest mutation rate population.

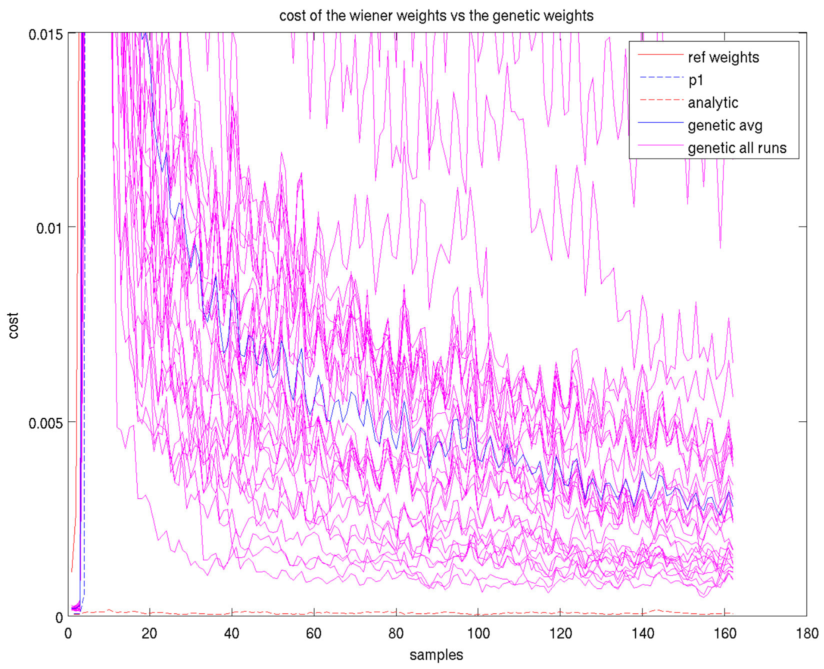


Figure 3.5: Simulation of genetic algorithm with 76 individuals in the lowest mutation rate population.

From figure 3.5, one can see that the average cost of the genetic algorithm for the last 20 samples seems to have decreased. The average cost of the last 20 samples is now around 0.003. The last 20 samples are used for comparison because this is the level the genetic algorithm are able to reach after some time of adapting to the big change in sample five. It is natural that it takes some generations before the genetic algorithm can find weights that cancel out the direction of the jammer source. Figure 3.6 shows the simulation with 64 individuals in the lowest mutation rate simulation.

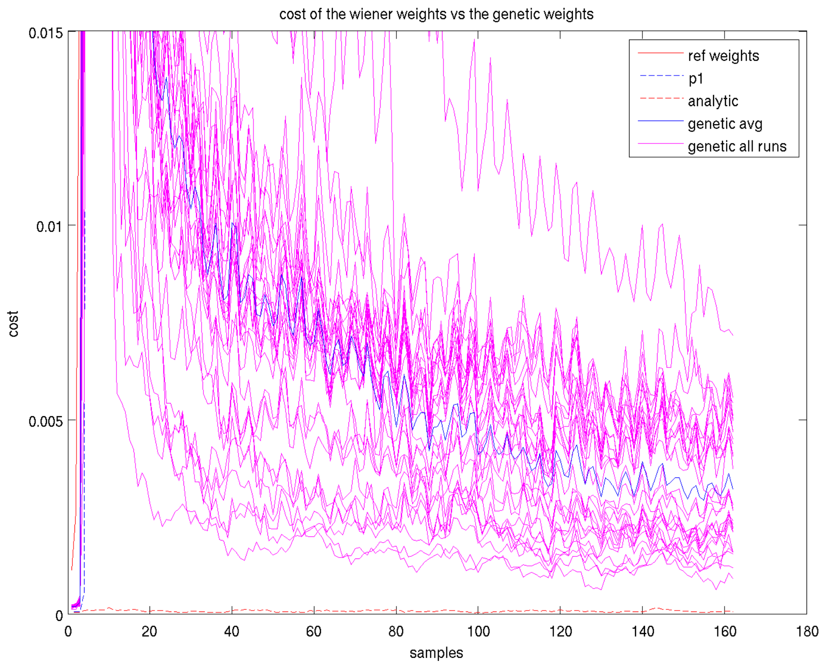


Figure 3.6: Simulation of genetic algorithm with 64 individuals in the lowest mutation rate population.

By decreasing the number of individuals in the main population further, the performance seems to be approximately the same. The number of individuals in the populations that do not have the lowest mutation rate have so far been five, eight and twelve. It seems like the small populations have been too small to be effective in the 85 individual scheme. This might cause the performance of the 76 individual scheme and the 64 individual scheme to be better. Figure 3.7 shows the simulation with 40 individuals in the lowest mutation rate population.

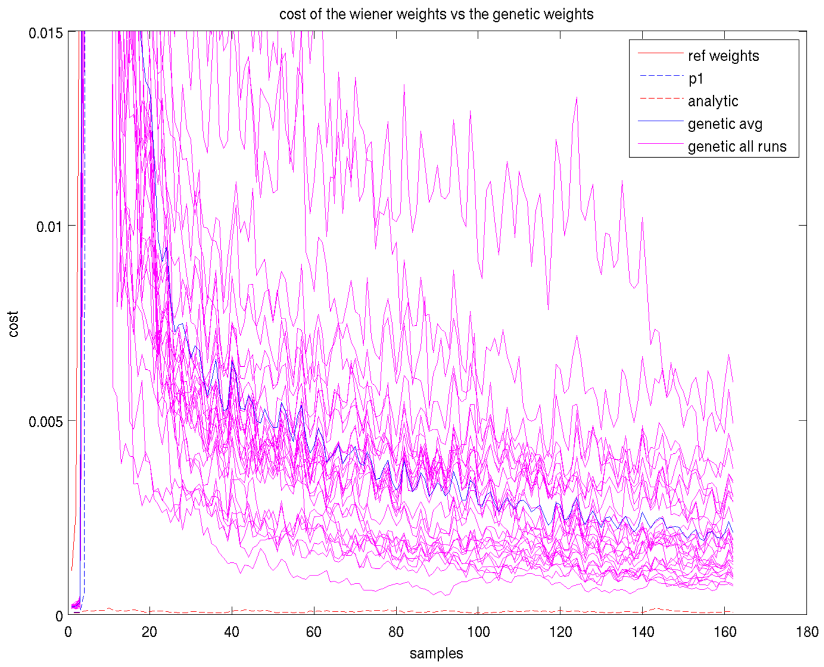


Figure 3.7: Simulation of genetic algorithm with 40 individuals in the lowest mutation rate population.

The average cost of the last 20 samples of the genetic algorithm weights seems to decrease further when the individuals now are even more evenly distributed. In figure 3.8 the simulation with evenly distributed individuals are shown.

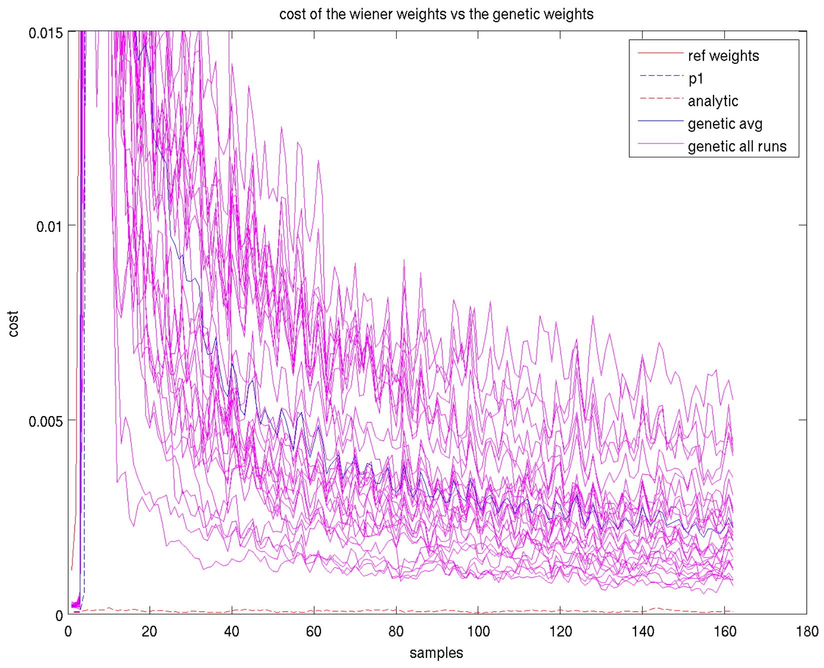


Figure 3.8: Simulation of genetic algorithm with 25 individuals in each population.

In figure 3.8 it can be seen that the cost of the genetic algorithm with evenly distributed individuals seems to be approximately the same as or better than the other simulated schemes. It can be observed that in all the figures of section 3.4, the cost of the weights of the different runs differ quite a lot. The averaging of 30 runs might not be sufficient to statistically conclude which of these schemes are the best for the sample data set. It is also simulated only a hand full of schemes. But because having evenly distributed individuals in the populations simplifies the hardware, and seem to have better than or equal to performance than the other simulated schemes, it is chosen to have evenly distributed individuals in hardware implementation.

3.5 Performance of algorithm

To measure the performance of the algorithm, the genetic algorithm is compared with both the DMI method and the non-preprocessed signal. The measure used is the power of the signal which is found by multiplying the weights with the antenna signal and summing them together. The weight of the reference signal is one. For the non-preprocessed signal the power of the reference signal is used. Figure 3.9 shows the power of the genetic algorithm signal compared with the average power of the DMI solution and the average power of the reference channel.

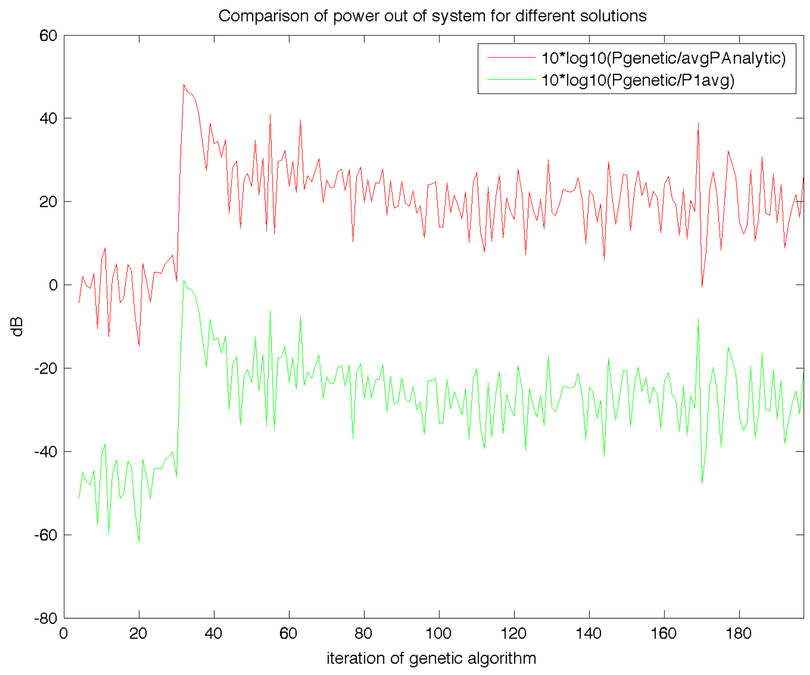


Figure 3.9: Comparison of output power for genetic algorithm, DMI and reference signal.

The green line in figure 3.9 shows the reduction in power received because of the genetic algorithm weights. The red line shows how much more the power can be reduced. In both lines it is the average power of the DMI and reference channel that the genetic algorithm weights are compared with. Reduction of received power of around 20 dB is definitively an improvement, but as one can see form figure 3.9 there is still room for a lot of improvement. The number of possible solutions are very large. But the genetic algorithm will find better and better solutions if more generations are allowed per data sample. Eventually the genetic algorithm will find the optimal Wiener solution. The number of calculations needed per generation makes one generation per sample unlikely with the target FPGA. This means that for the target FPGA the result is likely to be less than 20 dB reduction in received power.

Chapter 4

Hardware Design

This chapter will describe how the algorithm is implemented in the FPGA. For readability different alternatives are discussed and chosen throughout the chapter.

4.1 Top level architecture

The genetic algorithm might have many individuals and the processing of the individuals demand a lot of resources. Figure 4.1 shows the operations that is performed on each individual every generation.

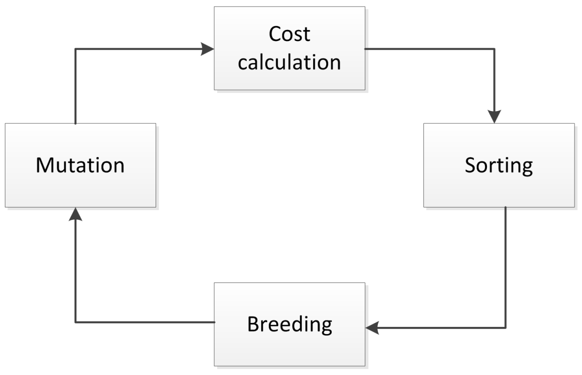


Figure 4.1: Operations in genetic algorithm.

The total processing time needed per individual per generation is critical for the performance of the algorithm. The combination of a lot of operands, complex operations and a as low as possible time consumption is challenging. The operations needed are too resource demanding to have dedicated cores for each individual when using the target FPGA.

There have to be some cores that is reused for different individuals.

One individual consists of nine complex weights. Each weight is a complex number with a ten bit real part and a ten bit imaginary part. This means that each individual consists of 180 bit. An example population of 100 individuals will consist of 18000 bits. With this amount of data, the data flow in the circuit is a challenge. A lot of data will need to be accessed from different operational cores. The first architecture considered were similar to a multi-core heterogeneous processor system. Figure 4.2 shows this suggested architecture.

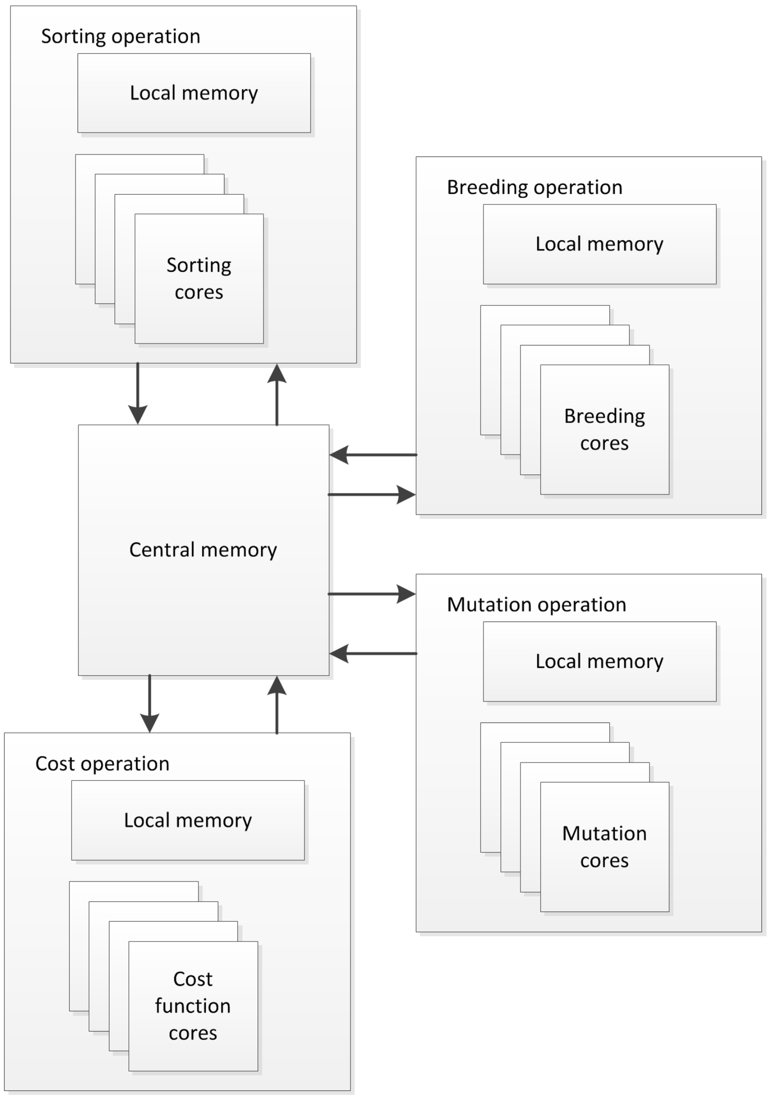


Figure 4.2: Heterogeneous core architecture.

As figure 4.2 shows one could have a central memory which holds the whole population and operation specific modules. Each operation module could have a local memory to hold one population. Each operation module could work on one population at a time. When all operation modules are finished writing the resulting populations back to the main memory, the populations can be sent to a new operation module.

The different operations will have different delays and latencies. This makes it difficult optimize the throughput of the design with a simple processor like architecture. The algorithm can be designed as a long pipeline where the throughput is the same in each operation core. An overview of a pipeline architecture is shown in figure 4.3.

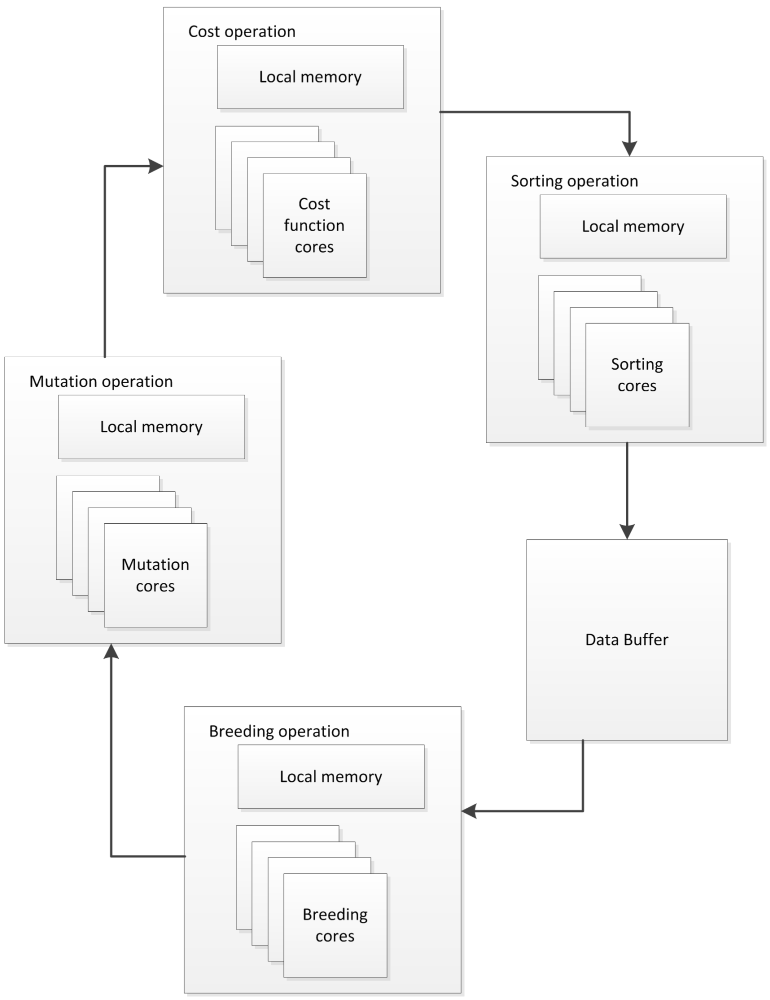


Figure 4.3: Pipeline architecture.

This architecture exploits that the order of the operations always are the same, such that the data does not need to go through a central memory between every operation. A data buffer is used to push data into the pipeline and receive data from the end of the pipeline. The idea is that this module controls the data flow and that all the other modules have the same throughput. This pipeline architecture is less generic compared to the more traditional multi-core heterogeneous architecture and can not easily be reused for other problems. The main challenge with the genetic algorithm design is the flow of all the data. The pipeline architecture have the advantage that the data flows in the same pattern every clock cycle such that there is less need for MUXes and alternative paths for the data. The more traditional architecture has a number of cores. The data have to constantly flow between these cores and the main memory. The amount of data flow to and from the main memory is a challenge. This challenge might be solved by streaming, broadcasting or priority. Common for all these schemes are that they are a trade off and that they come with disadvantages. The main advantage of the pipeline architecture is the reduced memory writing and reading. This saves time. The time consumption is critical for the performance of the algorithm. The pipeline architecture is therefore chosen for the implementation in this thesis.

4.2 Cost function

The main computational challenge of the algorithm is the computation of the cost function. The cost function, which was discussed in chapter 3, is shown in (4.1).

$$\text{cost} = p_1 - \mathbf{r}_{xx}^{\dagger} \cdot \mathbf{w} - \mathbf{w}^{\dagger} \cdot \mathbf{r}_{xx} + \mathbf{w} \cdot \mathbf{r}_{xd} \cdot \mathbf{w} \tag{4.1}$$

p_1, which is the average power of the reference channel for the last K samples, is a scalar. $\mathbf{w}$, which is the weights, is a 9×1 complex matrix. $\mathbf{r}_{xx}$, which is the auto correlation vector, is a 9×1 complex matrix. $\mathbf{r}_{xd}$, which is the cross correlation matrix, is a 9×9 complex matrix. An overview of the dimensions that is being multiplied together is shown in figure 4.4.

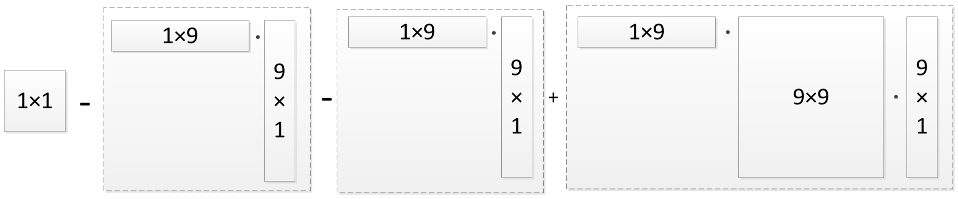


Figure 4.4: Dimensions of the cost function calculation.

The last term is a $(1\times9)\cdot(9\times9)\cdot(9\times1)$ calculation and can be divided into two parts. First a $(9\times9)\cdot(9\times1)$ calculation which results in a 9×1 matrix, then a $(1\times9)\cdot(9\times1)$ calculation which is the same operation as for the other non-scalars terms. In the $(9 \times 9) \cdot (9 \times 1)$ calculation the vector product of each of the rows of the 9×9 matrix with the 9×1

vector is found. For each vector product nine complex multiplications is needed. The final operation of the calculation of the $(1 \times 9) \cdot (9 \times 9) \cdot (9 \times 1)$ term as well as the calculation of the two other non-scalar terms is also vector products. This means that the calculations mostly consists of complex vector products with vector length nine. The resources in the FPGA are limited. The time consumption of the algorithm is critical for the performance of the design. There will always be a trade off between latency, delay and resource usage when it comes to circuit design. Finding a vector product module that is a good balance between performance and resource usage is crucial to the design.

4.2.1 Complex Multiplier

A complex multiplier is the building block for any complex vector product method. Multiplication of two complex numbers is shown in (4.2).

$$(a + ib) \cdot (c + id) = ac - bd + i(ad + bc) \tag{4.2}$$

In (4.2) i is used for $\sqrt{-1}$ and a,b,c and d are scalar real fixed point numbers. As one can see from (4.2), the complex multiplication consists of four real multiplications followed by two real additions or subtractions. Real multiplication can be performed in the FPGAs DSPs. As mentioned in chapter 2 the width of operands of the built in multipliers in the DPSs are 25×18. One real multiplication with operands with less width than this can be executed in one cycle by one DSP. Figure 4.5 shows the simple complex multiplier design.

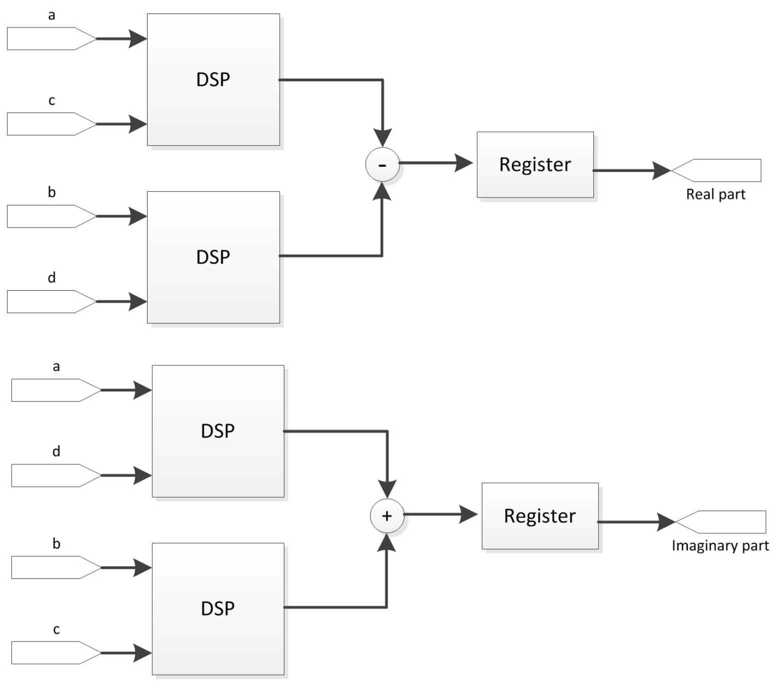


Figure 4.5: Complex multiplier design.

As figure 4.5 shows an output register is included in the design. This means that the output signal does not come with a computational delay. This design policy is used in all the modules of the design of this thesis. This complex multiplier is fully pipelined and has a latency of two clock cycles.

4.2.2 Complex Vector Product

When calculating the complex vector product using the complex multiplier from section 4.2.1, two alternatives were considered.

Fast and Pipelined Vector Product Module

A fast but resource consuming method to calculate a length n complex vector product is to use n complex multipliers and two tree adders. One for the real part and one for the complex part. This consept is shown in figure 4.6.

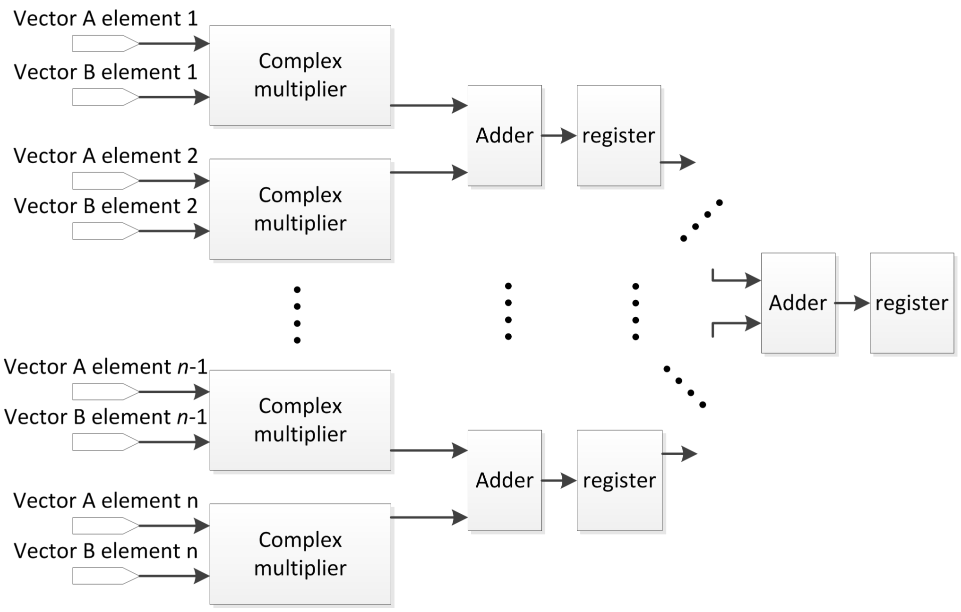


Figure 4.6: Fast and pipelined vector product module.

Figure 4.6 shows the concept of vector product calculation with a tree adder. Vector A and Vector B are complex vectors of length n. In the figure only one adder and one register is shown for each stage in the adder tree. It is also only drawn one line from each of the complex multipliers. As mentioned earlier it will be one tree adder for the imaginary part and one for the real part. The figure is simplified to be easier to understand. The adder tree can be pipelined by having memory for every sum as shown in the figure above. With a fully pipelined adder tree, new data can be set to the inputs of the adder tree every cycle. The complex multiplier is also fully pipelined, therefore the resulting vector product module can receive new input data on every clock cycle. The latency of an adder tree is $\lceil log_2(n) \rceil$. The total latency through this fully pipelined vector product module is $\lceil log_2(n) \rceil + 2$. The number of DSPs needed are four for each element of the vectors or $4 \cdot n$.

Slow and non-Pipelined Vector Product Module

The number of DSPs available in an FPGA is limited, to use one DSP for each of the multiplications in the vector product calculation might not be beneficial. Another method to calculate the vector product in hardware is to use two accumulators and less DSPs. The DSPs are used to calculate one part of the vector product on each clock cycle. This result of one part of the vector product is then added to the current results in the two accumulator. One of the accumulators holds the real part of the result and one accumulator holds the imaginary part of the result. When all the parts of the vector product are added in the accumulators, the result is valid. As shown in figure 4.7 one complex multiplier from section 4.2.1 can be used in combination with the two accumulators.

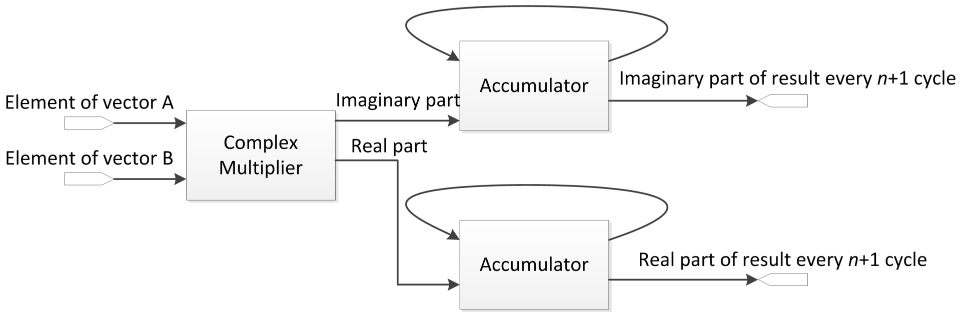


Figure 4.7: Slow and non-pipelined vector product module.

One could use one complex multiplier (four DSPs) and two accumulators for the vector product calculation, independent of the vector length n. The module is not pipelined and the delay of the module is $n+1$. New data can be put to the input of the circuit every $n+1$ clock cycles. $n+1$ clock cycles because n cycles is used to accumulate the result and one cycle is used to empty the accumulator.

4.2.3 Calculation of the Cost Function

To calculate the cost function as fast as possible and fully pipelined one can use multiple instances of the fast and pipelined vector product module. Figure 4.8 shows how the fast and pipelined vector product module can be used to implement the cost function.

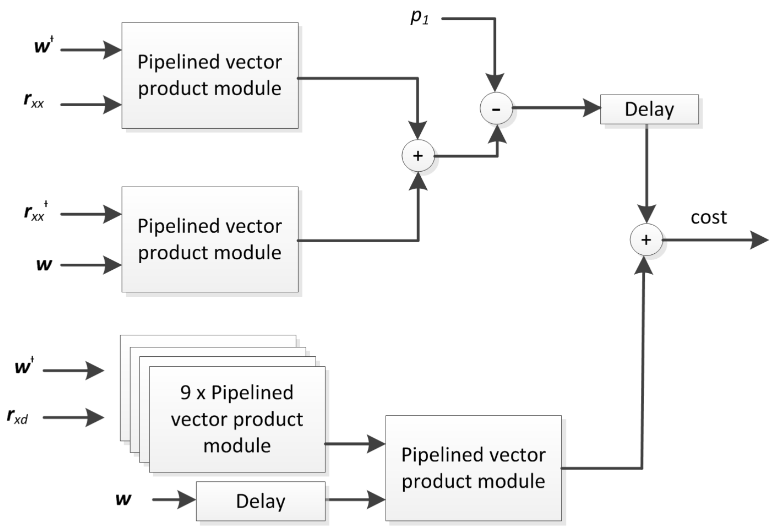
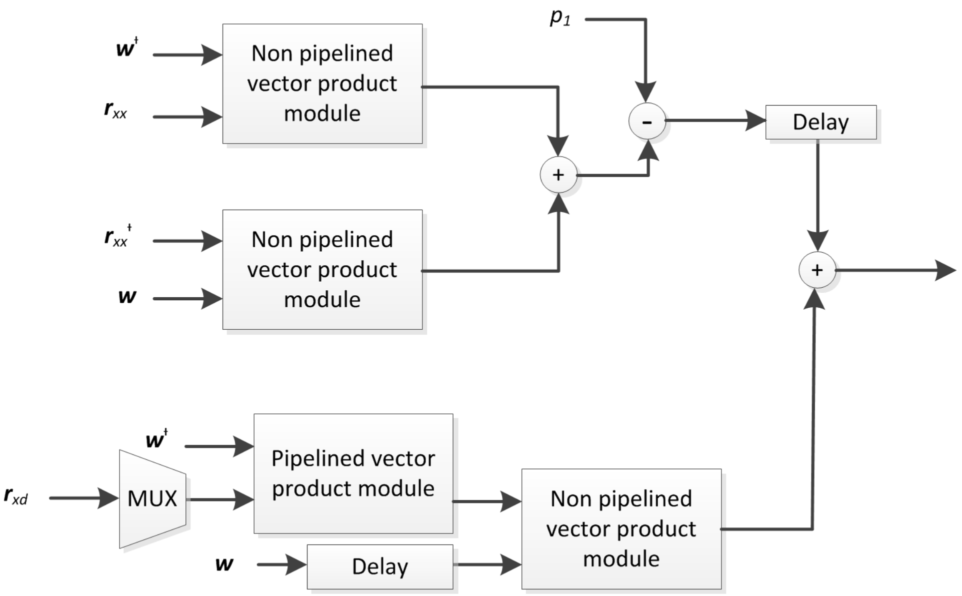

$w^{\dagger}$
r_{xx}
Pipelined vector product module
$r_{xx}^{\dagger}$
w
Pipelined vector product module
p_1
-
+
Delay
cost
+
$w^{\dagger}$
r_{xd}
9 x Pipelined vector product module
w
Delay
Pipelined vector product module

Figure 4.8: Fast and pipelined cost function module.

As one can see in figure 4.8, delay elements are used. The delay elements are used to balance the delays of the different calculations. This is done so that the cost function calculation can be fully pipelined. This method for cost function calculation uses $3 \cdot n + n^2$ complex multipliers, is fully pipelined and has a latency of $2 \cdot (\lceil log_2(n) \rceil + 2) + 1$.

A different method to calculate the cost is to use a combination of the slow and non-pipelined vector product module and the fast and pipelined vector product module. In the cost function the term $(w)^{\dagger} \cdot \mathbf{r}_{xd} \cdot \mathbf{w}$ is the most complex calculation. As mentioned earlier this calculation consists of n vector products that must be calculated first, then a vector product using the result of these n vector products. The other non-scalar terms consist of just one vector product. One could save resources, by using the slow and non-pipelined vector product module for the terms which is just one vector product. This means that the cost function module only can receive new input data every $n + 1$ clock cycles. A combination of the to different vector product modules can be used to calculate the most complex term of the cost function. Fast and pipelined multipliers can be used to calculate the first vector products and a slow and non-pipelined module can accumulate the result. Since the fast vector product module is pipelined and the slow and non-pipelined vector product module does not use all the elements of the input vectors at the same time. A combinations where less than n fast vector product modules can be used. One could for instance use one fast vector product module and one slow vector product module. This scheme is shown in figure 4.9.



$w^\dagger$
r_{xx}
Non pipelined vector product module
$r_{xx}^\dagger$
w
Non pipelined vector product module
p_1
-
+
Delay
+
$w^\dagger$
r_{xd}
MUX
Pipelined vector product module
w
Delay
Non pipelined vector product module

Figure 4.9: Slow cost function calculation.

This is a much slower and much less resource demanding method. $3+n$ complex multipliers are used for a cost function calculation with delay of $n+1$ and latency of $\lceil log_2(n)\rceil + n + 3$. A lot of resources is saved by making the cost function non-pipelined. One could use more than one of these non-pipelined cost function modules to make the delay smaller and the resource usage higher. It is a trade off between throughput and use of resources. By adjusting the number of complex multipliers per vector product module, the number of vector product modules per cost function module and the number of cost function modules, one could adjust delay and resource usage. For the specific implementation of this thesis the n is 9 and 640 DSPs are available. A fully pipelined implementation of the cost function will use $3 \cdot 9 + 9^2 = 108$ complex multipliers, with four DSPs per complex multiplier, this results in 432 DSPs. This is almost $\frac{3}{4}$ of the available DSPs in the target FPGA. Since the genetic algorithm module is only one part of the FPGA design and the rest of the system is not designed yet, it is not clear if so many DSPs can be used in the genetic algorithm module. One could create multipliers in the general logic of the FPGA, but this is resource demanding. Instead a cost function module with delay $9+1=10$ and latency $\lceil log_2(9)\rceil + 9 + 3 = 17$ with the use of $3+9=12$ complex multipliers as described above was implemented. One could use a number m of these cost function modules in parallel to make a throughput of $m \cdot \frac{1}{10}$ individuals per clock cycle. This makes it easier to adjust the resource usage of the circuit if the other parts of the design is dependent on a lot of resources. The final implementation of this thesis uses ten instances of this module, which give a throughput of one individual per clock cycle. This is the same throughput as for the fully pipelined implementation. If one compare the two implementations the fully pipelined implementation have less latency and use less resources. But the final implementation is easier to scale down and modify than the fully pipelined one and was therefore chosen. Figure 4.10 shows how the cost function calculation is implemented.

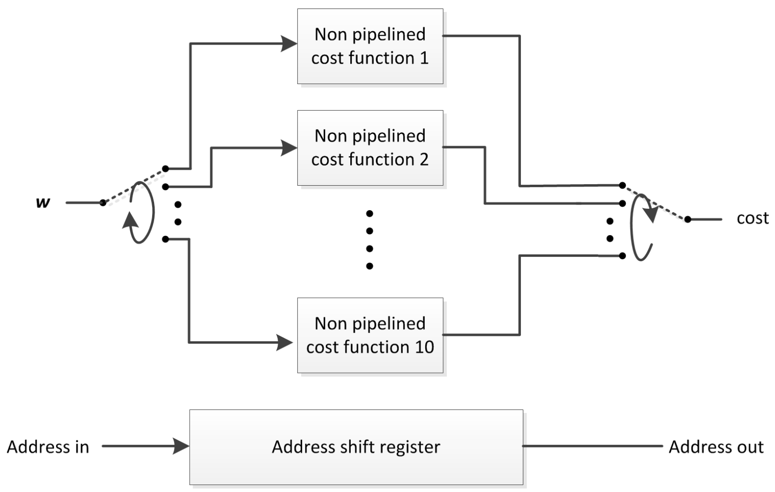

Non pipelined cost function 1
Non pipelined cost function 2
Non pipelined cost function 10
w
cost
Address in
Address shift register
Address out

Figure 4.10: Implemented cost function calculation.

As figure 4.10 shows one set of weights are sent to one cost function calculation module at a time. This is done to achieve not only a throughput of one individual per clock cycle, but an input rate and output rate of one individual per clock cycle. A constant data flow will simplify the flow of data in the other parts of the circuit. The thought is that one individual flows from each module to the next on every clock cycle. In the FPGA the modules will be spread and not have their own designated area. But since the calculations of the modules often are related they will, to some degree, be gathered. By having the same amount of data flow from each module to the next module on every clock cycle, the number of paths needed from one module gathering to the next module gathering is minimized. One can also see an address shift register in figure 4.10. This will be described in section 4.2.4

4.2.4 Flow of Individuals through Cost Function Module and Sorting Module

In the breeding module and the mutation module, the individuals are used and modified before they are sent to the next module. In the cost function module and the sorting module on the other hand the individuals are used to find the cost, before sorted and then sent to the next module. The individuals are not modified, and they are only used for the first $n + 1$ clock cycles of the cost function calculation directly. When calculating the cost, one needs to keep control of what individual the cost is calculated for. When the individuals are sorted with respect to their cost, knowing what cost corresponds to which individual is crucial. As seen in section 4.2.3 there is used an address shift register in the cost function. When the individuals arrive at the cost function module they are also sent to a block memory. The address in the block memory where the individual is put is sent into the cost function module. This address is delayed as many clock cycles as the latency of the cost function using an address shift register, then it is set on the output of the cost function module. This means that the cost of an individual is sent to the sorting module together with the address where the individual is located in the block memory. The addresses are then sorted with respect to their cost. Finally the sorted addresses are sent to the block memory and the corresponding individuals are sent to the next module. Figure 4.11 shows how the individuals,**w** , flow through the two modules.

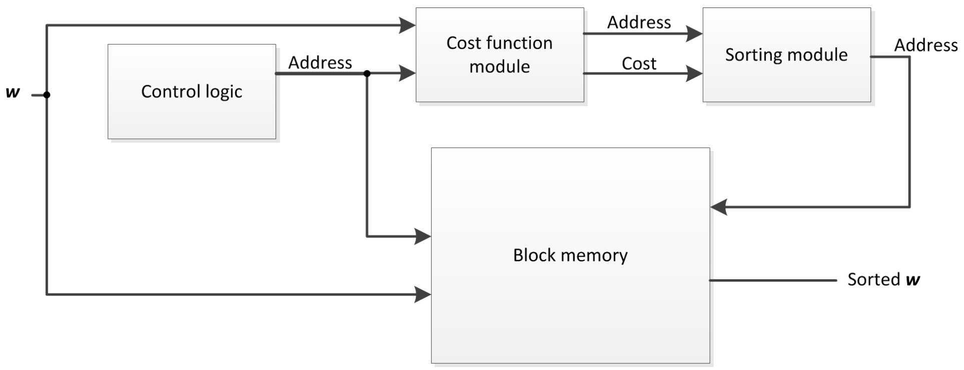


Figure 4.11: Implemented cost function calculation.

Another alternative would be to skip the block memory and to shift the whole individuals through the cost function. One would also need to keep the whole individuals together with the cost during the sorting. Let us assume that one uses a memory which can store all m individuals to avoid overwriting an individual before it is read after the sorting. Then the address length will be $\lceil log_2(m) \rceil$. For 65 to 128 individuals this results in an address length of seven bits. This is a very short word length compared to the 180 bit wide individuals. As mentioned the individual or address will have to be delayed 17 clock cycles to be synchronized with the cost after the cost function. To do delay a word of length W 17 clock cycles fully pipelined will take $W \cdot 17$ registers. This means 3060 registers if one use the individuals and 119 registers if one uses the addresses instead. There will be some extra routing and extra logic if one uses a block memory. There is also a latency when one uses a block memory. It usually takes one or two clock cycles from when a read signal is sent to the block memory, to when the result from that memory location is returned. Register usage is not the biggest problem of this design, but the savings are significant. The savings mentioned are just for the cost function, for the sorting it will also be savings in register usage. It might also be easier to route the address elements than the individual elements, simply because of the difference in width. Therefore the design of this thesis is implemented with a block memory, such that shorter addresses can be used in the cost module and the sorting module.

4.3 Sorting Individuals

After the cost of the individuals is calculated, the individuals or addresses of the individuals needs to be sorted. This needs to be done to be able to perform evolutionary processes on the populations. Because of the parallel nature of the FPGA, FPGAs are well suited for sorting. It is desired to have a sorting algorithm with an input rate and an output rate of one individual per clock cycle. This is as mentioned earlier done to simplify routing of the data flow between the modules. If one entire population is buffered, one could use one of many sorting algorithms well suited for sorting on an FPGA. It is however possible to sort on the fly using insertion sort in linear time on an FPGA. This will give the possibility to naturally receive one individual per clock cycle. The idea is, as always for the insertions sort algorithm, to insert every new element in the correct position in relation to the already sorted elements. It is several possibilities in how to technically fill an element into a list. One could use pointers. The pointer of the last element is updated to point at the element that is filled in. The element one fills in is set to point to where the last element was already pointing. This is shown in figure 4.12.

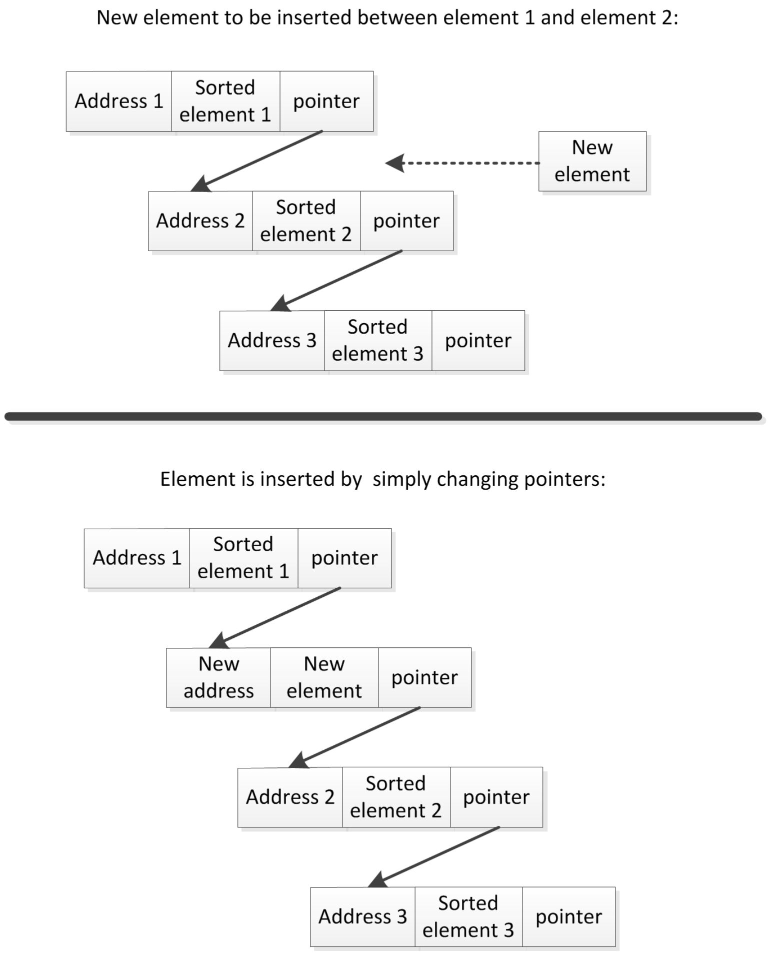


Figure 4.12: One sorting element.

The pointer would for instance be the memory address where the next element is placed. This is a typical solution in software. In an FPGA a shift register inspired method can be used. The elements which have larger cost that the new element can be physically shifted to make room for the new element. This concept is shown in figure 4.13.

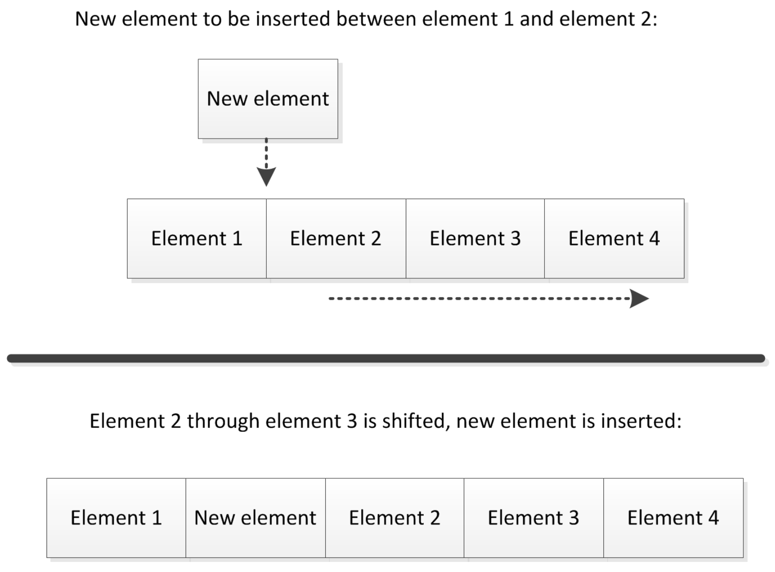

New element to be inserted between element 1 and element 2:
New element
Element 1
Element 2
Element 3
Element 4
Element 2 through element 3 is shifted, new element is inserted:
Element 1
New element
Element 2
Element 3
Element 4

Figure 4.13: Connection of sorting elements.

Comparators and MUXes are used to decide what element is going to go into each sorting element on every clock cycle. A function diagram for what should go into one sorting element is shown in figure 4.14.

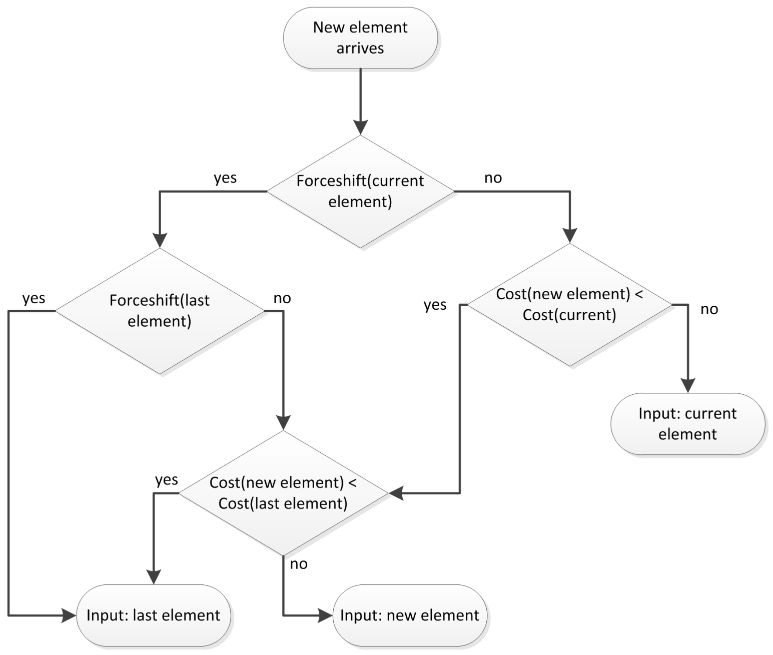


Figure 4.14: What the sorting element should be next.

As mentioned this is realised by a combination of MUXes and comparators. The force shift signal in figure 4.14 comes from a shift register. This shift register, called the force shift register, is used to transmit the result of the sorting. The force shift register is also used to force the sorted elements out of the sorting element. When a sorting is complete the highest cost individuals address is located in the last sorting element. The lowest cost individuals address is located in the first sorting element. The force shift register is one bit wide and have the same length as the number of elements to be sorted. The force shift register is filled with ones when each sorting is finished. Each of the bits in the shift register is connected to one of the sorting elements force shift input. For each clock cycle a zero is shifted into the force shift register. As long as a sorting element have a one at the force shift signal it can not keep its address. The next address will as seen in figure 4.14 come from either the last sorting element or the new element to be sorted. When filling the force shift register with ones, a new sorting will begin which is no longer connected to the old elements. The output of the sorting module is connected to the last sorting element. When a sorting is done all elements are force shifted out. This means that the output will be the elements sorted from high cost to low cost. One address of one individual per clock cycle. Figure 4.15 shows the process of emptying and filling the sorting elements.

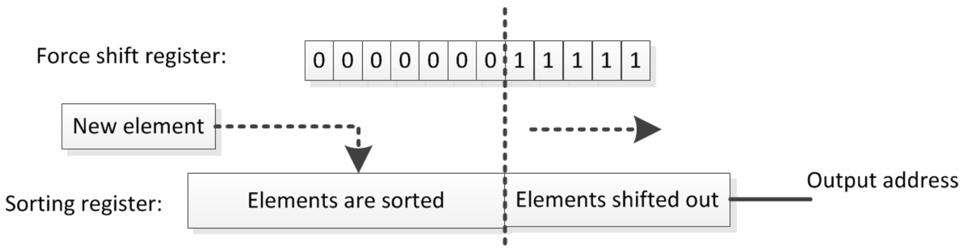


Figure 4.15: The sorting process.

As long as one wants to have all the elements sorted, the length of the force shift register and the number of sorting elements must be the same as the number of elements to be sorted. The output element will always be a valid element, as long as a new sorting is started every time a sorting is finished. If one however does not need all the elements sorted. One could simply lower the number of sorting elements and the length of the force shift register. The last bit of the force shift register can be used to tell when the data is valid. In the specific design of this thesis, half the population is discarded after sorting to make room for new individuals after breeding. There is no point in sorting elements before discarding them. This means that the number of sorting elements can be half as many as there is individuals in one population. This saves around half the hardware of the sorting process and simplifies routing and timing. If one assumes a population of an even number m individuals. The result is that the sorting algorithm transmits the $\frac{m}{2}$ best individuals, highest cost first, for $\frac{m}{2}$ clock cycles. Then elements that is not good enough to make the next top $\frac{m}{2}$ are sent in the next $\frac{m}{2}$ clock cycles. This have a m clock cycle period.

4.4 Data Buffer

The idea of the pipeline architecture of this algorithm is that it is fully pipelined to enhance operational speed. This means that it can receive one individual per clock cycle and that it finishes processing one individual every cycle. If the latency, L, of the pipeline had been the same as the number of individuals, m, the finished elements could start a new processing cycle at once. But since this is not always the case a memory buffer is needed. This can be solved by a 180 bit wide shift register with length $m - L$. This works if the delay of the pipeline is not changed.

Another possibility is to make a more complex data buffer. One could have a block memory that can hold all the individuals. This buffer could send individuals in the rate best suited for the breeding module. A new population signal could be sent every time the first individual of a new population is sent. This new population signal would then flow through the other modules and back to the buffer again. This signal can then be used to know where in the memory the received data should be placed. Because it is a block memory there is a delay between when signals are set to read data and when the data is ready from the block memory. The breeding module, which is the next module, uses two individuals at a time to produce children. The breeding module transmit both the two parents and the two children to the next module. The breeding module should have an output rate of one individual per cycle. This means that two new parents are needed by the breeding module every forth clock cycle. The data buffer therefore transmits in this rate to the breeding module. The data buffer also handles migration. Instead of transmitting the two worst individuals of each population, the two best individuals from the last population is transmitted.

This data buffer is a more complex solution which include counters, MUXes and a block memory. However, as long as the total number of individuals is higher than the delay of the pipeline, this data buffer module is independent of the delay of the pipeline. This means that changes in the pipeline usually does not effect the design of the data buffer. The data buffer just pushes data into the pipeline and receives at the other end when the data is valid. The first population is assumed to be the main population, which usually has low mutation rate. When the best individual of the first population is received in the data buffer, this individual is set to the output of the design. Figure 4.16 sums up the operations of the data buffer module.

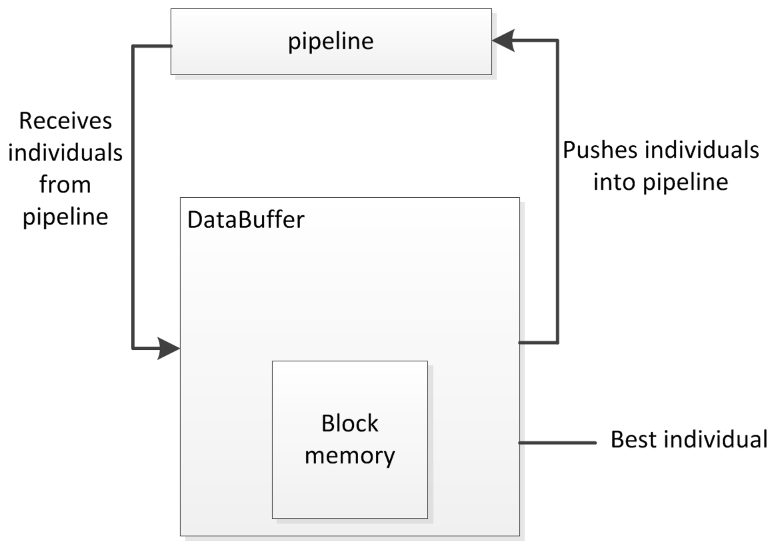

pipeline
Receives individuals from pipeline
Pushes individuals into pipeline
DataBuffer
Block memory
Best individual

Figure 4.16: Data buffer oprations.

4.5 Breeding and Mutation

Breeding and mutation can be realised in simple hardware. Breeding of two individuals per clock cycle and mutation of one individual per clock cycle, can be realised by a reasonable amount of MUXes and LUTs. One problem however is that a lot random bits are needed to perform these processes. This is especially a problem for the mutation. If one individual should be mutated with a mutation rate of $\frac{1}{128}$, seven random bits are needed per bit of the individual. This means 1260 random bits per clock cycle. It is challenging to generate that many random bits. There are several strategies to generate random bits, but generating that many random bits every clock cycle demands a lot of resources. In general the more random the bits need to be, the more effort is needed to generate them. It is assumed that the genetic algorithm module can receive this amount of random bits on every cycle from surrounding logic. The problem of randomness and the degree of randomness needed in the algorithm are severely complex topics. The focus of this thesis have been implementing the algorithm itself. A solution for how to generate this amount of random bits is therefore not presented in this thesis.

4.5.1 Breeding

The breeding module receives two individuals, or parents on every fourth clock cycle. During this period both the parents and the two children are transmitted, one individual on each clock cycle. The one of the parents which is assumed to be the best parent is sent first, then the other parent, then the two children. Each bit of the children is randomly selected from one of the two parents corresponding bits. A new population signal is sent together with the best parent after a new population signal is received from the data buffer.

4.5.2 Mutation

The mutation module receives one individual per sample and transmits one individual per sample. The mutation module takes mutation rates of the different populations as inputs. When a new population signal is received the mutation rate is changed and the first individual is not mutated.

4.6 Simulation and Verification

Most of the modules in the design of this thesis have the same test structure, which will be described later in this section and will be called standard test structure. Table 4.1 shows an overview of how the different modules are tested.

Module	**Description**	**Test status**
$GeneticAlg$	The entire design	Visual testing for random data
$DelayDataBuffer$	The data buffer	Visual testing for random data
$Breeding$	The breeding module	Standard test structure
$Mutation$	The mutation module	Standard test structure
$SerialCostFunction$	10 cost function cores	Standard test structure
$CostFunction$	The cost function module	Standard test structure
$ComplexHVecMatrixVecMul$	Calculates the hermitian conjugate of vector multiplied with matrix multiplier with vector	Standard test structure
$ComplexVectorMul$	Fast vector product module	Standard test structure
$ComplexAdderTree$	Part of fast vector product	Not separaty tested
$ComplexVectorMulSlow$	Slow vector product module	Standard test structure
$ComplexMul$	Complex multiplier	Standard test structure
$SortingRegister$	The	Standard test structure
$InsertionSort$	Insertion sort module	Standard test structure
$SortingRegister$	Shift register used in the insertion sort	Not tested separatly
$Accumulator$	Generated from Xilinx coregen	Not tested separatly
$Blockmemory$	Generated from Xilinx coregen	Not tested separatly

Table 4.1: Testing of the different modules.

4.6.1 Modules Generated using Xilinx Coregen

Both the $Accumulator$ module and the differently sized $Blockmemory$ modules are generated using Xilinx software called Coregen. It has not been prioritized to test these modules separately. They are all part of tests which include modules that instantiate the coregen modules.

4.6.2 Modules Tested with Standard Test Structure

As mentioned most of the modules of this design have been tested using the same test structure. Procedures have been written in VHDL to simplify testing. These procedures take one set of data that the module is to be tested for. The procedures then drives the module with this set of data. The correct output of this set of data is then calculated and compared with the output from the module. If the output does not match the expected output, warnings are sent out from the test bench. By making such procedures for the modules, it is simple to test the modules with any input data. The modules are all tested with loops that apply random data to the module. Some of the modules have also been specifically tested for corner cases.

4.6.3 Visual Testing of Modules

For the $DelayDataBuffer$ module and the $GeneticAlg$ module there have been performed visual testing. It is seen in the waveforms that the circuit operates as expected. The cost is gradually decreasing and increase only when the input is changed. The individual that is sent into the data buffer comes out at the correct time.

4.7 Synthesis and Place and Route

The genetic algorithm of this thesis is to be part of an FPGA design. The genetic algorithm module have a lot of inputs; auto correlation, cross correlation, random bits and p_1. The genetic algorithm can, because of its number of inputs and output, not be routed by itself on the target FPGA. The other parts of the FPGA design have not been available for the work done this spring. Therefore the genetic algorithm had to be wrapped before it was synthesised. Figure 4.17 shows how the design is wrapped to be able to do synthesis on the genetic algortihm design.

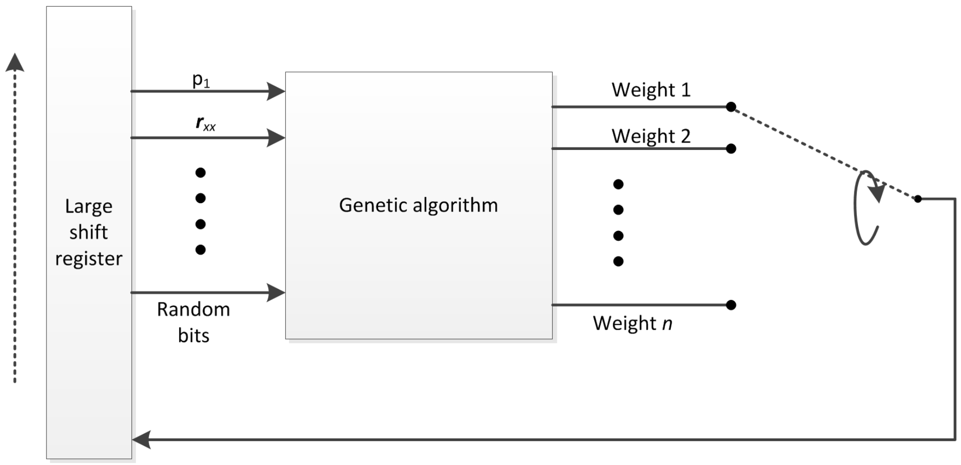


Figure 4.17: Wrapping the genetic algorithm module.

As one can see from figure 4.17 a large shift register is used. The shift register receives data from the different outputs of the genetic algorithm module in turn. The shift registers values are used as input for the genetic algorithm module. This is done to make sure that the input data to the genetic algorithm module are random. Another advantage of this design is that the outputs of the genetic algorithm module are being used. If the inputs are not varying or the output is not used, parts of or the whole module might be simplified. This will make the synthesis results worthless. To synthesize the module by itself, when it is going to part of a bigger FPGA design, might give unrealistic results. But the parts of design that will be challenging to synthesize might be reveled. A limit to how fast the module can be run on the target FPGA is also found.

Under place and route, routing soon became a problem in the design. This was especially for the cost function where a lot of data is used in the DSPs. The first place and route results had a maximum clock cycle frequency of around 80 MHz, which is far from the 200 MHz goal. Path delay was the main problem, a lot of data were going to the same location. The cross correlation and auto correlation matrices are at most changed every generation. It was therefore allowed for this change to take three clock cycles. The shift register used to simulate the inputs to the genetic algorithm module might give the module an extra challenge compared to if the inputs were coming from a module. The inputs from a different module might be less dependant on each other. Timing problems for all the inputs of the genetic algorithm module were therefore ignored. This will create even more ideal conditions for the genetic algorithm module. Again the goal is to find a limit of how fast the genetic algorithm module can be run and not the exact speed of which it can run. By ignoring the timing problems of the input signals the maximum frequency of the synthesised module increased to around 130 MHz. To further increase the clock frequency rate at which the genetic algorithm module can be run, one could delay difficult paths with registers as shown in figure 4.18.

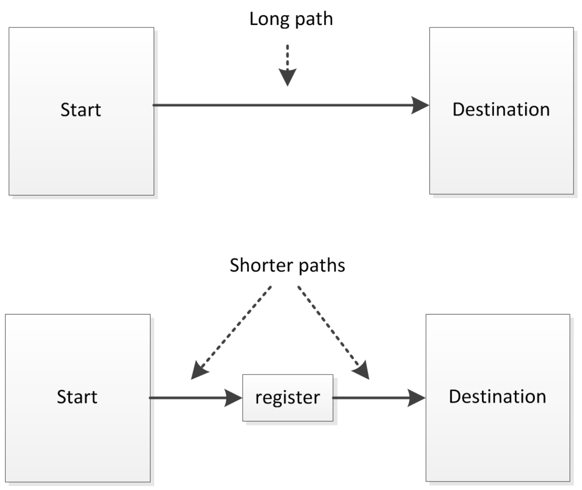


Figure 4.18: Splitting path with register.

When path delay is the main problem, as it was for this design, this is the main technique to improve clock frequency if not redesigning the module itself. By using this technique the max clock frequency of the place and route was increased to 180 MHz(Appendix A).

Chapter 5

Discussion

The discussion is divided into two section. Section 5.1 contains the general discussion regarding the parameters of the algorithm and the algorithms performance compared to the DMI method. Section 5.2 contains the general discussion regarding the hardware implementation of the algorithm.

5.1 Algorithm for Implementation

There is still a lot of work that can be done on improving the algorithm in this thesis. As explained in chapter 3 it has only been attempted to find good parameters for the algorithm and not the optimal parameters. The number representation of the weights in the implementation limits the size of the weights. This actual limit used in the implementation does not effect the simulation results because the optimal weights for the sample data set is not bigger than this limit. This limit is not general. To determine how large weights are needed, knowledge about the specific antenna system is needed. This includes information about antenna geometry, sampling and reception hardware and specifications of what the antenna system should handle.

5.1.1 Performance

For the measuring of the performance of the algorithm, it is used a generation frequency equal to the sample rate. This is not a reasonable generation frequency. As seen in chapter 4 the achieved throughput of the hardware design of this thesis is one individual per clock cycle. For a population of 100 individuals and a clock cycle rate of 200 MHz this would mean a generation rate of 2 MHz. With a typical sample rate of 100 MHz of a GPS antenna system. The generation rate should be 50 times lower than the sample rate. This means that the result of one generation of the genetic algorithm needs to be used on 50 samples. In the performance simulation the result from each generation of the genetic algorithm is used on only one sample. This will effect the resulting performance of the algorithm. The algorithm is run 50 times as fast as it can be run in the hardware design compared to the sample rate. This means that the measured performance is likely to be a lot better than the actual performance will be.

For performance comparison the DMI is used. The DMI method finds the optimal weights for K samples. As for the genetic algorithm it is not reasonable to run the DMI once per sample, as was done in the performance simulations. The time consumption of the DMI algorithm is even bigger than generation period of the genetic algorithm. This means that the performance of the DMI algorithm in the performance simulations is better than it should be. Both the performance approximation of the DMI and the genetic algorithm are invalid when it comes to actual performance of the system. However, when comparing the performance of the genetic algorithm to the DMI method, the actual relation between them can be assumed to be better than or equal to what the simulations have shown.

5.2 Hardware Design

A student is not an experienced hardware designer. The different alternatives compared in the hardware design chapter might therefore not be the absolute best alternatives. It has been attempted to look at other designs and find good solutions to the problems that have been encountered and to compare them. As the design have been run through place and route one have seen that some of the solutions in the the design have worked well and other have not.

5.2.1 Bottleneck

Under synthesis it is one data path that have been the limiting factor for the maximum frequency of the design. This is the data path for the individuals sent from the mutation module to the cost function module. Figure 5.1 shows the data path where there have been routing problems.

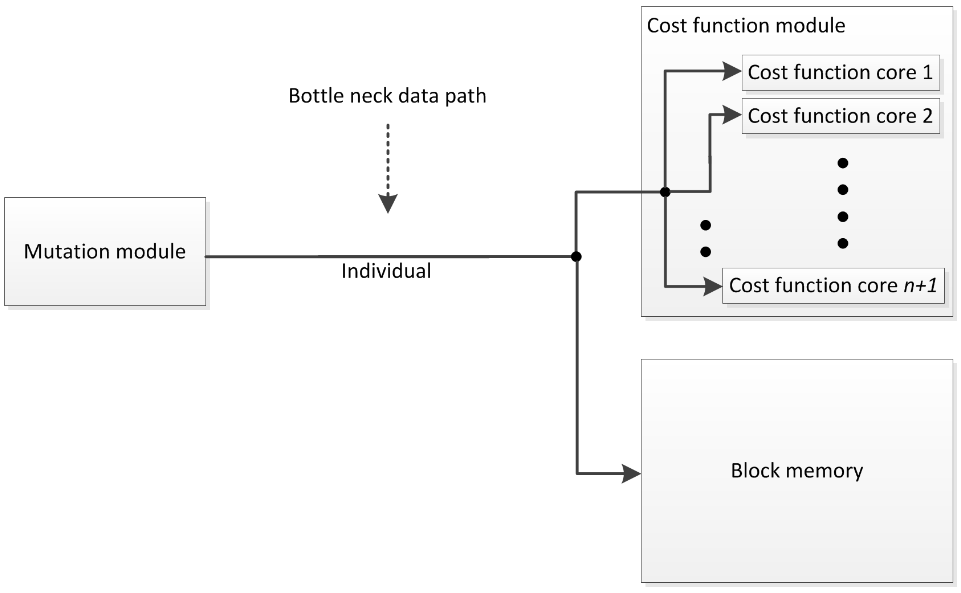


Figure 5.1: Bottle neck data path.

As one can see from figure 5.1 the individuals that come from the mutation module are connected to both the block memory and the cost function module. Figure 5.1 also shows that the individuals are connected to all the $n + 1$ cost function cores. Remembering that one individual is 180 bit wide, this is a lot of connections. Figure 5.2 shows the attempt to solve the problem.

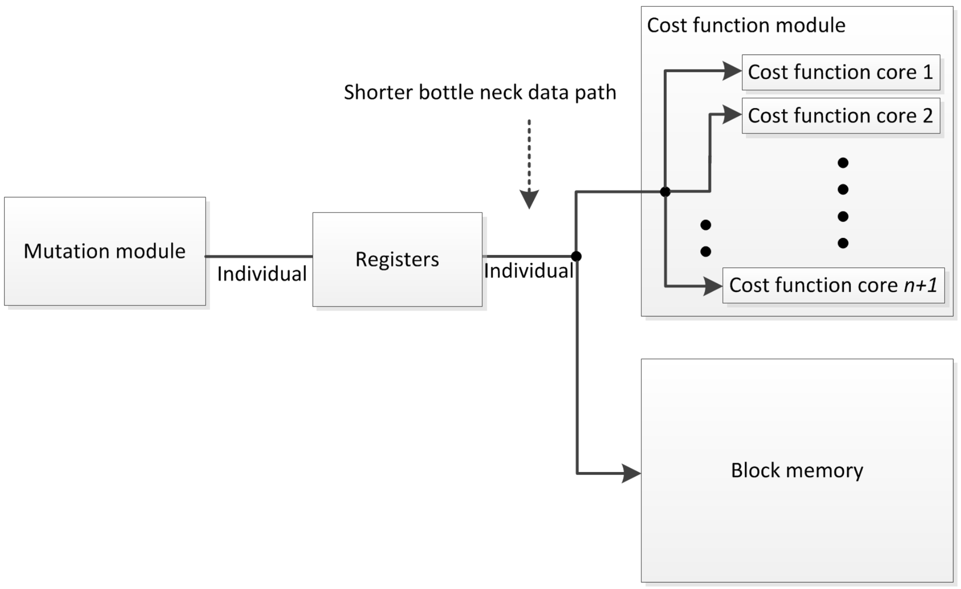


Figure 5.2: Bottle neck data path, divided by registers.

As described in the hardware design chapter, inserting register elements in long data paths can reduce the length of each data path. However, if too many registers are inserted it is difficult to route because of the number of paths and registers. For each register step inserted one clock cycle of latency is also added to the path.

To realise a pipelined cost function two alternatives were discussed. One were to use multiple slow cost function cores with a delay of ten clock cycles. The other were to use one fast and fully pipelined cost function module. The main argument for using multiple cost function cores was scalability. The fast cost function module solution will for the realised throughput of one individual per clock cycle save both DSPs and LUTs. The fast cost function module solution might use some extra registers because all the data paths have to be balanced to the same latency. In resource usage the two solutions are similar and they both have advantages and disadvantages. When it comes to their impact on the data path of the individuals coming in to the module, this was not considered when the design choice was made. It is not unlikely that the fully pipelined cost function module, that simply takes one set of weights to the same DSPs on every clock cycle, might be easier to route effectively than the more complex input of the multiple cost function core solution.

Another alternative is to use the scalability of the chosen cost function module to reduce the throughput of the module. This means less bits that need to be connected to the cost function module. Less bits to connect means easier routing, so this could simplify the problem. One could also experiment more with putting registers into the longest data path. In this thesis it has only been attempted to put the same amount of registers into the same place of all these long paths. It might be beneficial to place the registers different places in the different paths to avoid that the paths are so similar. When paths are similar, the best hardware path is more likely to be the best hardware path for many of these paths.

5.2.2 Verification

The verification of the design is not finished. It has not yet been written good test benches for the top level design and the main data buffer. Although all the modules that is instantiated in the top level design is tested, one can not be sure that all parts work well together just by visual inspection.

The procedures produced for the tested modules are easy to use and the modules can easily be tested with more random data, corner cases or fully covered. A lot of work was put into generating these self testing test benches. When changes are made to the design it is very convenient to be able to verify the design without much extra work. As long as inputs and outputs of the modules are not changed, the test procedures allow for the modules to be tested without visually inspecting waveforms or changing test bench code.

It has not been found time to do hardware testing of the design. Two options were considered. One could physically build an antenna system and design all the other modules needed to test the design. One could also use the available test data set, stream the data from a computer into the FPGA, run the data through the genetic algorithm design and stream the results out to a computer. The computer could then be used to look at the results and verify that this is the same results as under simulation. Figure 5.3 shows how this can be done.

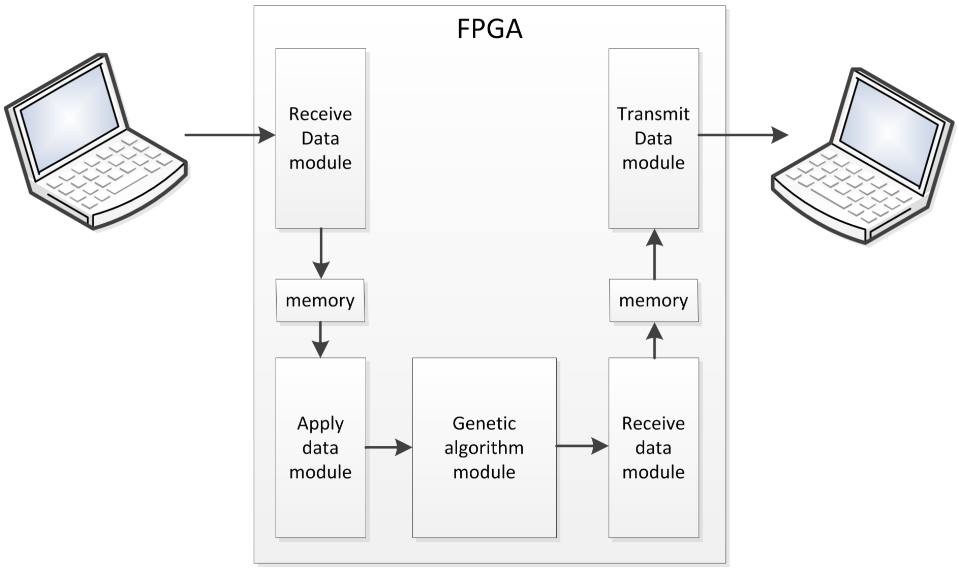


Figure 5.3: Hardware testing using a computer.

Such a test would not give much more coverage than the VHDL simulations. The testing of the module in a real antenna system can give a lot more coverage. Since the main goal of this thesis is to see if the genetic can replace the DMI algorithm. The verification of the design have not been the main focus. If the design is used in a real antenna system, the design should be more thoroughly tested and verified first.

5.2.3 Performance

The performance of the design is basically one individual per clock cycle of throughput and a maximum clock frequency of 180 MHz. Can this performance be increased?

The throughput of the circuit of one individual per clock cycle is quite good when one remember how many calculations one individual needs for each generation. For instance the 109 complex multiplications and the effort needed to sort one individual for each clock cycle. The genetic algorithm module itself is using around 75% of the available DSPs and maximum 17% of the LUTs and the registers. This means that there is a lot of resources left in the FPGA. These resources could have been used to improve the throughput of the design. One have to remember that the target FPGA will be used for more than just the genetic algorithm module. It is also already difficult to reach the 200 MHz maximum frequency goal.

The nature of the algorithm with a lot of data flow is a challenge. This challenge will only increase if it is attempted to increase the throughput of the design. The memory architecture chosen for this design will become more complex if the throughput is increased. More than one individual need to be per clock cycle. The pipeline architecture used is clearly well suited for the throughput of one individual per clock cycle. If the throughput is changed, the more traditional memory architecture might be better suited. In general the more traditional memory architecture, with one main memory and multiple different cores for the operations, are easier scaled than the pipeline architecture. This might make it easier to reach a specific maximum frequency goal for the circuit. On the other hand, it will be hard to make a more traditional design with a throughput of one individual per clock cycle or more on the target hardware.

5.2.4 Scalability

The design is scalable by constants in the main VHDL package. The constants set the number of antennas, taps per antenna, populations and individuals per population. The width of the data types used for the individuals in the VHDL code is also set from the main package. The idea was to make the design scalable such that only the cost function module and the data types had to be changed to make a genetic algorithm that calculated a different problem. Writing generic code might reduce hard coded optimization that could have been done if the actual numbers were used instead of constants. But such optimizations are usually done by the synthesis tool anyway.

The scaling of the number of antennas and taps per antenna is done simply by reducing or increasing the number of multipliers in the cost function and the number of resources used in the other modules. This is a very simple way of scaling the design and this will only work as long as there is enough resources in the FPGA to do the calculations and as long as the design is possible to route. If the number of antennas and taps per antenna is scaled down the design becomes ineffective and a lot of resources will be unused. If the number of antennas and taps per antenna is scaled up the design might be too resource demanding or not possible to route. A better way of scaling the design would be preferable.

The scaling of the number of populations and number of individuals per population is only possible within the size of the block memories which are set in the design. These can easily be replaced if the memory needed exceeds the memory available in the block memories. The total number of individuals also have to be larger than the latency of the pipeline.

As mentioned the chosen vector multiplier scheme sacrifices resource usage and route ability for scalability. The memory architecture sacrifices scalability for throughput.

Chapter 6

Conclusion

A genetic algorithm has been implemented in VHDL, synthesised and partially verified. The algorithm performs adaptive nulling and reduce the power of the output. When implementing the algorithm trade offs between scalability, max frequency, throughput and flexibility have been made. The genetic algorithm design needs surrounding logic which is not in the scope of this thesis. This surrounding logic have not yet been designed, so the genetic algorithm module have been synthesised by itself. A maximum clock cycle rate of 180 MHz has been achieved for place and route, which is simplified because the essential surrounding logic of the final FPGA design is not included. The 200 MHz clock cycle rate goal has not been achieved, but changes to the design have been discussed which might increase the maximum clock cycle rate.

In hardware design it is always trade offs. In this particular design a lot of choices have been made. Both choices of parameters for the algorithm and different opportunities for the hardware design have been discussed. In this thesis, the algorithm have just been limited to genetic algorithm. The hardware design have just been limited to the target FPGA. This gives a lot of freedom. With so many design and parameter choices, it is not attempted to find the absolute best solution. But it is shown, that an FPGA implementation of a genetic algorithm can be used in a real time system. It is also achieved an approximately 20 dB increase in SNR for the sample data set, although this was done with a too high generation rate. The genetic algorithm can not find weights close to the optimal solution found by the DMI algorithm in real time. There is no question to the potential of the genetic algorithm used in a non-real-time system. The genetic algorithm can run faster than the DMI algorithm, but can for this problem not realistically meet the performance of the DMI in real time. The number of possible solutions is simply to wast.

Chapter 7

Future Work

It is now shown that the algorithm can be implemented on an FPGA. The performance of the algorithm can be improved by finding the optimal constellation of populations with different mutation rates, sizes and migration schemes for the specific problem to be solved. One can use information of the actual antenna system to determine the best number representation of the weights. More accurate performance simulations can be done by including the time consumption of the algorithms. The time consumption of the genetic algorithm was found in this thesis. An FPGA implementation of the DMI method can be made to find the time consumption of the method. The more precise simulations will give actual performance. The results can be used to more accurately compare the solutions.

The hardware design can be further improved to reach desired clock cycle rate of 200 MHz. The hardware design verification can also be improved. A solution to the problem of the number of random bits needed and the required randomness of this bits needs to be found. The surrounding modules needs to be developed before a hardware test can be run.

Bibliography

[1] Chen, P. Y., Chen, R. D., Chang, Y. P., Shieh, L. S., Malki, H. A., 2008. Hardware implementation for a genetic algorithm. IEEE Transactions on Instrumentation and Measurement 57 (4), 699–705.

[2] Kreyszig, E., 2006. Advanced engineering mathematics (9th edition). John Wiley & Sons, Inc.

[3] Mitchell, M., 1998. An Introduction to Genetic Algorithms. MIT Press.

[4] Monzingo, R. A., Haupt, R. L., Miller, T. W., 2011. Introduction to Adaptive Arrays (2nd edition). SciTech Publishing.

[5] Myrick, W. L., Zoltowski, M. D., 1999. Anti-jam space-time preprocessor for gps based on multistage nested wiener filter. Military Communications Conference Proceedings 1 (1), 675–681.

[6] Rosado, A., Iakymchuk, T., Bataller, M., Wegrzyn, M., 2008. Hardware-efficient matrix inversion algorithm for complex adaptive systems. The 19th IEEE International Conference on Electronics, Circuits and Systems(ICECS) 1 (1), 41–44.

[7] Tang, W., Leslie, Y., 2004. Hardware implementation for a genetic algorithms using fpga. The 47th IEEE International Midwest Symposium on Circuits and Systems 1 (1), 549–552.

[8] Xilinx Inc., 2012. Virtex-6 family overview.
URL `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`

Appendix A

Electronic Appendix

The VHDL code, MATLAB code and the timing report of the final FPGA design is located in the electronic appendix for this thesis. The electronic appendix contains three folders. One for each of the three mentioned categories.

A.1 VHDL Code Folder

In the VHDL code folder there is two folders. One called simulation and one called source.

A.1.1 Simulation Folder

The simulation folder contains all simulation entity files which ends with ”ent.vhd”, all simulation files which ends with”sim.vhd” and a package consisting of ComplexPrivateTb_bdy.vhd and ComplexPrivateTb_pck.vhd. The package contains all simulation procedures and simulation constants.

A.1.2 Source Folder

The source folder contains all the VHDL source code for the final design. Each module is divided into one entity file ending with ”ent.vhd” and one structural level description file ending with ”str.vhd” or a rtl level description file ending with ”rtl.vhd”. Figure A.1 shows how the different modules are used in the final design.

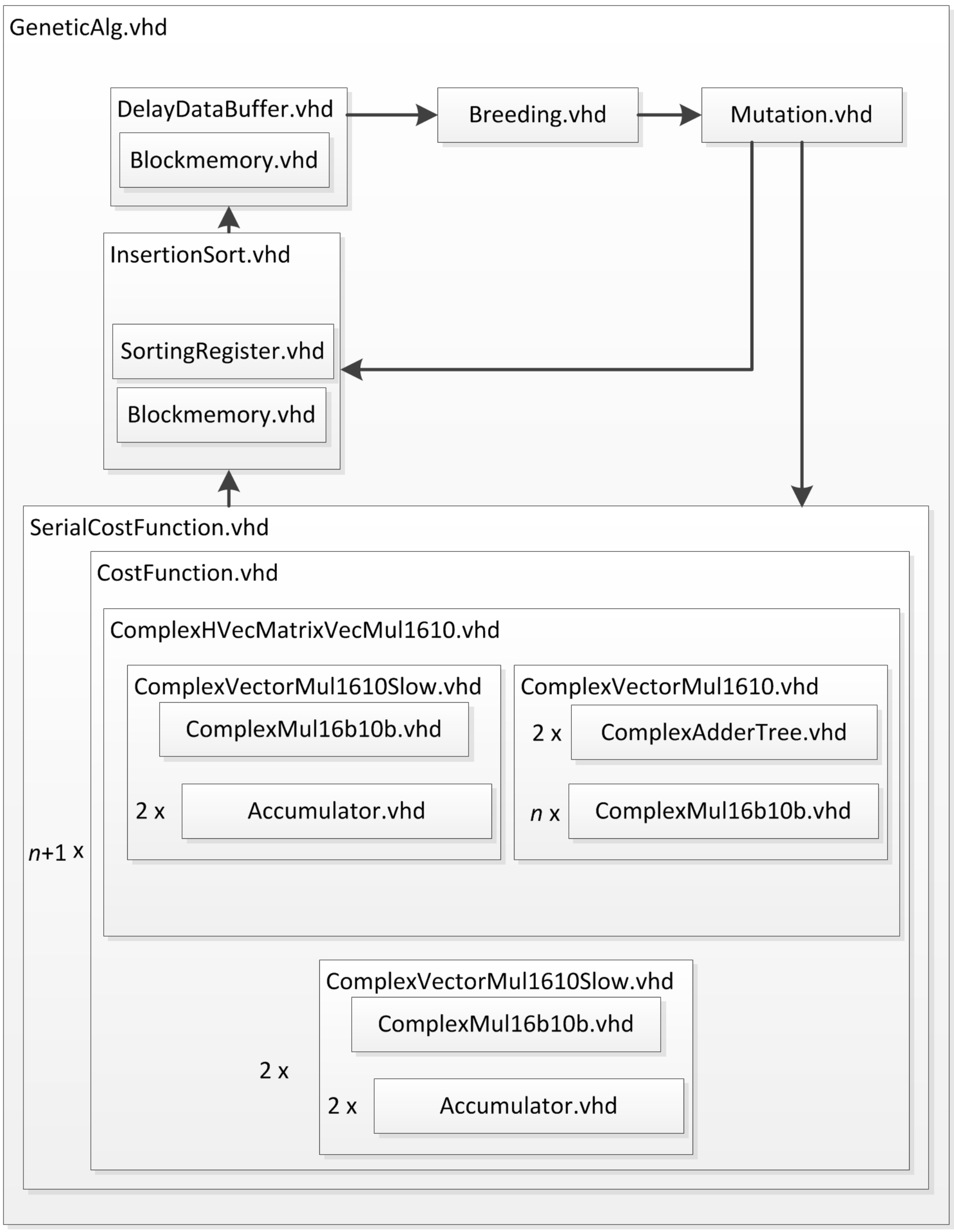


Figure A.1: Structure of VHDL code.

In figure A.1 the module names does not include the endings ”ent.vhd”,”rtl.vhd” and ”str.vhd”. The ”Blockmemory.vhd” name is simplified.

A.2 MATLAB Code Folder

The MATLAB code folder contains all the MATLAB code developed for this project. The MATLAB code folder includes a input generation folder. The input generation folder contains all the MATLAB files provided by Kongsberg Aerospace & Defence. These MATLAB files generates the input for the generic algorithm module.

A.3 Timing Report Folder

In the timing report folder the timing report for the final design can be found. This report shows a minimum period of 5.534 ns. This corresponds to a maximum frequency of approximately 180 MHz.