UNIVERSITY OF CINCINNATI

Date: 01/18/2008

I, Harish Ramaswamy,
hereby submit this work as part of the requirements for the degree of:
Masters of Science

in:
Computer Engineering

It is entitled:

An Extended Library of Hardware Modules for Genetic Algorithms,
with Applications to DNA Sequence Matching

This work and its defense approved by:

Chair: Dr. Carla Purdy
Dr. Harold Carter
Dr. Robert Ewing

An Extended Library of Hardware Modules for Genetic Algorithms, with Applications to DNA Sequence Matching

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

Masters of Science

in the Department of Electrical and Computer Engineering
of the College of Engineering

January 2008

by

Harish Ramaswamy

B.E. in Electrical and Electronics Engineering
College of Engineering, Guindy, Anna University, India

Thesis Advisor and Committee Chair: Dr. Carla Purdy

Abstract

Genetic Algorithms (GAs) are intelligent search techniques that are based on Darwin's theory of evolution. They are a particular class of Evolutionary Algorithms (EA) which can be applied to find acceptable solutions to computationally complex search problems in a reasonable amount of time. GAs are usually implemented in software and typically require a large number of iterations to reach a near optimal solution, thus consuming a large amount of processing time. The inherent parallelism in GAs facilitates the exploration of a large number of regions in the search space and also becomes the motivation for their implementation in hardware.

As a part of this thesis, we construct a comprehensive library which implements commonly used techniques employed in the various stages of a GA – selection, crossover and mutation. We also provide modules to generate high quality pseudorandom numbers, which the GA needs to use to ensure high performance. The purpose of constructing the library is to maintain a varied repository of components that can be efficiently combined as needed to build larger architectures, depending on the specific application. Our project extends earlier work which implemented a smaller set of GA modules, along with a family of feedback based shift registers for pseudorandom number generation. In addition, we do a comparative analysis of the effectiveness of various module choices.

We also develop an architecture from our components for DNA sequence alignment. Our results are compared with results from ClustalW, a commonly used software application developed by the European Bioinformatics Institute (EBI) for this problem.

Acknowledgements

I would like to thank my advisor Dr. Carla Purdy for her support and guidance throughout my thesis work. I would also like to thank Dr. Carter and Dr. Ewing for spending their invaluable time reviewing my work and agreeing to be on my thesis committee.

I am grateful to my family and friends for their constant moral support and encouragement. Special thanks to Ashthy, Giri, Bisku and Thiaga and my roommates Haran, Vicki and Arvind for being there for me always and for making this journey extremely enjoyable.

Finally, thanks to the Bishop street guys for making UC a memorable experience.

Table of Contents

List of Figures	iv
List of Tables	vi
1. Introduction	1
1.1 Motivation	2
1.2 Introduction to Genetic Algorithms	2
1.3 Introduction to the Sequence Alignment Problem	3
1.4 Organization of the Thesis	4
2. Overview of Genetic Algorithms	6
2.1 Elements in a Genetic Algorithm	7
2.2 Simple Genetic Algorithm	8
2.3 Steady-state Genetic Algorithm	10
2.4 Previous Work on Hardware Genetic Algorithms	12
2.5 Previous Work on Software based Genetic Algorithms	14
2.6 Previous Work on Developing this Library	15
3. Genetic Algorithm Modules and Architecture in Hardware	16
3.1 Random Number Generators	17
3.1.1 Cellular Automata	17
3.1.2 Multiple Cellular Automata	19
3.1.3 Hybrid Cellular Automata	19

3.1.4	Speed-Area Comparison between LFSR and CA	21
3.2	Selection Module	23
3.2.1	Tournament Selection	23
3.2.2	Rank Selection	25
3.2.3	Truncation Selection	27
3.3	Reproduction Module	28
3.3.1	Half Sibling and Clone	29
3.4	Summary of GA Library Modules	30
3.5	Sample Architecture	31
4.	DNA Sequence Alignment	36
4.1	Introduction	36
4.2	Motivation	38
4.3	GA and Sequence Alignment	39
4.4	Previous Work	39
4.5	Application of our GA Library to Sequence Alignment	42
5.	Performance Analysis	44
5.1	Experimental Setup	44
5.2	Functional Verification	46
5.3	Performance Analysis	47
5.4	Results of DNA Sequence Alignment	60
5.5	Area / Timing Report	64

6. Conclusions and Future Work	67
6.1 Conclusions	67
6.2 Future Work	68
 Bibliography	 69

List of Figures

Figure 2.1	Steps Involved in a Genetic Algorithm	8
Figure 2.2	Flowchart for a Simple Genetic Algorithm	9
Figure 2.3	Flowchart for a Steady-state Genetic Algorithm	11
Figure 3.1(a)	CA with Null Boundary Condition	18
Figure 3.1(b)	CA with Cyclic Boundary Condition	18
Figure 3.2	Serial Test Result for 1D CA RNG	18
Figure 3.3	Serial Test Result for 32 CA RNG	19
Figure 3.4	State Diagram of a 498 Site Hybrid CA for 800 Time Steps	20
Figure 3.5	RC Representation of a Signal	21
Figure 3.6	Structure of a LFSR	22
Figure 3.7	Structure of a CA	22
Figure 3.8	Chart Showing Probability of Selection of Members According to Roulette Selection	26
Figure 3.9	Chart Showing Probability of Selection of Members According to Rank Selection	27
Figure 3.10	Chart Showing Probability of Selection of Members According to Truncation Selection	28
Figure 3.11	Block Diagram Representation of HSAC	29
Figure 3.12	Block Diagram of the Architecture Developed	32
Figure 3.13	Overall Architecture of the Hardware Genetic Algorithm (PHGA)	34
Figure 3.14	Neural Network Architecture	35

Figure 4.1	Types of Mutations	37
Figure 4.2	Block Diagram of GA Architecture in Hardware for MSA	40
Figure 5.1	Architecture of the GA System Implemented in this Work	45
Figure 5.2	Comparison of Results Between LFSR and Multi-LFSR Based GA Implementations for Function $f(x)=2x$	50
Figure 5.3	Comparison of Results Between LFSR and Multi-LFSR Based GA Implementations for Function $f(x) = 2x^3 - 45x^2 + 100x$	53
Figure 5.4	Results with Various PRNGs for Function $f(x)=2x$	55
Figure 5.5	Results for Various Selection Operators for Function $f(x) = 2x$ with Hybrid CA and Half-Sibling and Clone	58
Figure 5.6	Number of Instances of % Confidence Interval (90%) that includes ClustalW Value	64

List of Tables

Table 3.1	State Transition according to Rule 30	17
Table 3.2	Various Performance Metrics for LFSR and CA	22
Table 3.3	Example Population for Tournament Selection	24
Table 3.4	Fitness and Associated Rank for a Sample Population	26
Table 3.5	Summary of the Existing and Newly Added Modules in the GA Library	30
Table 3.6	Comparison of Performance Results between HEGA and PHGA	35
Table 4.1	Comparison of PGA with ClustalW	41
Table 5.1	Performance Results with Various Crossover Operators for Function $f(x) = 2x$	48
Table 5.2	Performance Results with Various Crossover Operators for Function $f(x)=2x$	49
Table 5.3	Performance Results with Various Crossover Operators for Function $f(x) = 2x^3 - 45x^2 + 100x$	51
Table 5.4	Performance Results with Various Crossover Operators for Function $f(x) = 2x^3 - 45x^2 + 100x$	52
Table 5.5	Performance Results with Various PRNGs for Function $f(x)=2x$	54
Table 5.6	Performance Results with Various PRNGs - Function $f(x)=2x^3-45x^2+100x$	55
Table 5.7	Performance Results with Various Selection Operators for Function $f(x) = 2x$	57

Table 5.8	Performance Results with Various Selection Operators for Function $f(x) = 2x^3 - 45x^2 + 100x$	59
Table 5.9	Sequence Alignment Results for Patterns of Length 50 (Each Case Repeated 10 Times)	61
Table 5.10	Sequence Alignment Results for Patterns of Length 40 (Each Case Repeated 10 Times)	61
Table 5.11	Sequence Alignment Results for Patterns of Length 30 (Each Case Repeated 10 Times)	62
Table 5.12	Sequence Alignment Results for Patterns of Length 20 (Each Case Repeated 10 Times)	62
Table 5.13	Sequence Alignment Results for Patterns of Length 10 (Each Case Repeated 10 Times)	63
Table 5.14	Average Difference between ClustalW and GA Values for 10 Sets (Positive Difference Indicates Better Performance by ClustalW)	63
Table 5.15	Area Report when Implemented on Altera FPGA Devices	65
Table 5.16	Speedup over ClustalW for Different Data-sets of Varying Length	66

Chapter 1

Introduction

Genetic Algorithms (GAs) are search algorithms based on the mechanics of natural selection and genetics. GAs were originally developed by John Holland, his colleagues, and his students at the University of Michigan in 1975 [1], and have been applied to many hard optimization problems including VLSI layout optimization problems [2], Boolean satisfiability [32] and the Hamiltonian circuit problem [3].

The basic GA consists of the following modules: data encoding, selection, crossover, mutation and fitness evaluation. The data encoding and fitness definition modules require problem specific information. However the other GA modules can be implemented more generally. Hence, a general purpose GA engine requires only certain modules to be changed according to the application, making it ideal for FPGAs and other reconfigurable systems. This thesis deals with the implementation of a library of GA modules in a hardware description language (VHDL). A sample architecture has also been developed, incorporating the principles of pipelining and parallelization, and has been applied to the DNA sequence alignment problem. The results have been compared with those of a software counterpart, CLUSTALW [39]. The selection, crossover and mutation modules developed are generic and hence can be used by other researchers and employed in various architectures to solve any optimization problem.

1.1 Motivation

Genetic algorithms (GAs) have been recognized as robust general-purpose optimization techniques that are capable of finding good solutions to hard optimization problems. But even on a high-performance machine, a software GA applied to a realistic complicated problem can run for days. This is especially true for any non-trivial application of a GA where the search space is really large. Hence the implementation of GAs on hardware would be an ideal alternative to problems too complex for software based GAs. Pipelining, parallelization and no function call overhead give a hardware implemented GA superior performance in terms of speed and efficiency when compared to its software counterparts.

This thesis involves the design of a comprehensive library of GA modules in VHDL which can be directly synthesized and mapped onto any target technology. The modules are also incorporated into a sample pipelined architecture which has been applied to the sequence alignment problem for DNA.

1.2 Introduction to Genetic Algorithms

GAs are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology. They are classified as global search heuristics. They differ from classic optimization techniques such as hill climbing or simulated annealing, in the following ways [1]:

- GAs use encoding of parameters and not parameters themselves
- GAs work on a population of points, not one unique point

- GAs use only the values of the function to be optimized, not derived values or any auxiliary knowledge
- GAs use probabilistic transition functions and not deterministic ones.

The mechanics of a basic genetic algorithm are surprisingly simple. The algorithm starts with an initial population of randomly generated solutions in the search space. Traditionally, the population members are represented as binary strings, but other types of encoding are also possible. Each member has a fitness value associated with it, specific to the application. Two members are randomly selected from the population to perform reproduction. The higher the fitness of the member, the greater is the probability of the member being selected for reproduction. The selected members reproduce, usually by a method called crossover, and undergo mutation, and the resulting offspring, which form the new population, are used in the next iteration. A GA uses the mutation operator to prevent the algorithm from converging to local optima. This process is repeated until the termination condition is met, which could be the number of iterations or saturation in fitness improvement.

1.3 Introduction to the Sequence Alignment Problem

Sequence alignment is one of the most important and basic concepts in computational genomics. It is the process of comparing two or more sequences with each other, with the purpose of highlighting the similarity between them. In other words, it is the process of determining the parts of a sequence that match with parts of another sequence. Sequence alignment techniques are important in the study of DNA, RNA and proteins. Sequence

alignment can be extended to deal with more than two sequences. The basic assumptions in sequence alignment are [4]:

- Sequences that are similar share a common ancestral sequence (sequences that share common ancestors are called homologous.)
- Due to common ancestry, similar sequences have similar functionality

Very short or very similar sequences can be aligned by hand. However, most realistic problems require the alignment of lengthy and highly variable, multiple sequences that cannot be aligned solely by human effort. A number of computational algorithms have been developed or adapted for this problem, including dynamic programming, genetic algorithms and other heuristic approaches. These methods, however, require long computation times for large sequences. Our hardware modules in the GA library have been adapted and incorporated into an architecture for the sequence alignment problem.

A number of software/hardware implementations have already been developed for this purpose [5, 6, 7]. Our work involves applying the general-purpose GA modules to find solutions to this problem and hence demonstrate the robustness and effectiveness of the library we have developed.

1.4 Organization of the Thesis

The thesis is organized as follows. Chapter 2 presents the different modules in a GA and the explanation of a simple and steady-state GA. It also talks about the previous hardware and software implementations of GAs. Chapter 3 deals with the modules that have been

implemented as a part of this work. It also explains the application chosen and the architecture that has been developed to solve the problem. Chapter 4 talks about how the GA library implemented as a part of this work can be applied to DNA sequence alignment. Chapter 5 analyses the results and compares our solution with the existing ones. Chapter 6 concludes by presenting a summary of the work and possible avenues that could be explored in the future.

Chapter 2

Overview of Genetic Algorithms

GAs are adaptive methods used to solve search and optimization problems. They are based on the genetic processes of biological organisms, i.e., populations evolve over many generations according to the principles of natural selection and “survival of the fittest”, first stated by Charles Darwin in his *Origin of Species* [33]. By mimicking this process, GAs are able to “evolve” solutions to real world problems, if they are suitably encoded [8].

In nature, individuals in a population compete with each other. The individuals that are most successful in surviving and attracting mates have relatively large numbers of offspring. Poorly performing individuals will produce fewer or no offspring at all. This means that genes from a highly “fit” individual will spread to an increasing number of individuals in successive generations. The combination of good characteristics from different ancestors can sometime produce “superfit” offspring, whose fitness is greater than that of either parent. In this way, species evolve to become more and more suited to their environment [8].

The power of GAs comes from the fact that the technique is robust and can be applied to a wide range of problem areas. GAs are not guaranteed to find globally optimal solutions, but they are good at finding “acceptably good” solutions to problems “acceptably quickly”. But where specialized techniques exist to solve particular problems, they are bound to outperform GAs in both speed and accuracy.

2.1 Elements in a Genetic Algorithm

GAs traditionally work on a *population*, which is an abstract representation of candidate solutions. Each member in the population is called a *chromosome*, analogous to the chromosomes in natural systems. The member chromosomes are usually encoded as bit strings and the individual character in each string is called a gene. Encodings other than bit strings, e.g., real numbers, can also be used. Initially many individual solutions are randomly generated to form the initial population. This population evolves over generations until the termination condition is met.

Each member in the population has a fitness value associated with it. The fitness value represents the quality of the individual solution and the fitness function is always problem specific. To form a new population (next generation), individuals are *selected* based on their fitness. Many selection procedures are currently in use, the most common being Holland's [9] original *fitness-proportionate selection*. The selected individuals undergo *crossover* and *mutation*. Reproduction is usually through crossover, which is performed between the two selected individuals called *parents*, by exchanging parts of their genome to form new individuals, called *offspring*. This operation tends to move the GA forward and explore the wide sample space for better solutions. The *mutation* operator is introduced to prevent premature convergence to local optima. It is usually carried out by flipping bits at random from the chromosome, with a small probability.

GAs are stochastic iterative processes that are not guaranteed to converge. The termination condition is usually specified as a fixed number of iterations (generations) or

saturation in fitness improvement. The steps involved in a genetic algorithm are shown below [Figure 2.1]:

Figure 2.1 Steps Involved in a Genetic Algorithm [10]

Genetic Algorithms are classified into two types based on the replacement scheme:

- Simple or Generational Genetic Algorithm
- Steady-state Genetic Algorithm

These are described in detail below.

2.2 Simple Genetic Algorithm

The simple GA, otherwise called the generational GA, involves complete replacement of the population between generations. The GA process starts with an initial population, which could be user supplied or randomly generated. The fitness of each member is evaluated depending upon the application. This is followed by the selection operator where two parents are selected for reproduction. The selection is based on the fitness, with a more fit individual having a higher probability of reproduction. The selected parents undergo crossover and mutation, producing offspring. This process is repeated

until an entire new generation is formed and this generation replaces the old population.

The whole process is represented as a flowchart in Figure 2.2.

Figure 2.2 Flowchart for a Simple Genetic Algorithm [35]

2.3 Steady-state Genetic Algorithm

The steady-state GA is also called the partial replacement GA because only a fraction of the existing population is replaced in an iteration. The functioning of a steady-state GA is very similar to a simple GA.

The steady-state GA starts with an initial population of randomly generated or user-supplied candidate solutions. From the population, pairs of chromosomes are selected as parents either randomly or based on fitness. These parents undergo crossover and mutation to produce offspring. These offspring now replace a portion of the existing population. The replacement scheme could be any of the following:

- Replace the worst
- Replace the best
- Replace at random
- Replace the parent

The steady-state algorithm is a better model of what actually happens in nature. In long-lived species like mammals, the offspring and parents live concurrently. This allows parents to nurture and teach the offspring, but also gives rise to competition between them [8]. The steps involved in a steady-state GA are shown in Figure 2.3.

Figure 2.3 Flowchart for a Steady-state Genetic Algorithm [34]

The essential difference between a conventional, generational GA and the steady-state GA is that, in the steady-state GA, the population statistics have to be computed after each mating, and the offspring are available immediately for reproduction. In other words, this type of a GA provides opportunity to exploit a promising individual as soon as it gets created. However no evidence has been found to prove that the steady-state GA is fundamentally better than the conventional simple GA [11].

2.4 Previous Work on Hardware Genetic Algorithms

The past several years have witnessed a sharp rise in work on hardware based genetic algorithms. All the research work done in implementing a GA on hardware is aimed at achieving speedup over the software counterparts by exploiting the principles of pipelining and parallelism and minimizing the logic resources used for implementation. This section presents a brief overview of the various hardware implementations of GAs carried out by researchers in the recent past.

Scott et al. described a fully functional hardware based genetic algorithm [12] in VHDL, designed to act as a coprocessor with the CPU of a PC. It was implemented on a prototyping board called BORG, which contained five Xilinx FPGAs. The design exploited parallelism and coarse-grained pipelining and was applied to problems where a conventional GA was slow.

Nedjah and Mourelle proposed an overall architecture for hardware implementation of a genetic algorithm. The fitness evaluation component was implemented on a neural

network [13] making the entire architecture problem independent, generic and massively parallel. The architecture was programmed into SPARTAN3 Xilinx FPGA [15] and was found to be more than 5 times faster than the SGA developed by Scott et al [12], but it occupied twice the area.

Graham and Nelson constructed a hardware genetic algorithm for optimizing the symmetric traveling salesman problem. The SPGA (Splash 2 Parallel Genetic Algorithm) [14] was implemented on Splash 2, a reconfigurable computer consisting of 4 FPGAs (Xilinx 4010) and associated memories. The system attained a 6 – 10 times speedup over its software counterparts. It was based on a simple GA.

Koonar et al. [16] proposed an architecture for implementing a genetic algorithm for circuit partitioning in VLSI. The design employed six modules and three external memories. It was implemented in VHDL, simulated using ModelSim and synthesized on Virtex part xcv50e using Xilinx ISE 4.1. It achieved over 100x improvement in processing speed when compared to the software implementations.

Wakabayashi et al. developed a new GA Accelerator (GAA) [17] where the crossover operator to be applied to each individual was dynamically selected during algorithm execution. The GAA was implemented using Verilog HDL, synthesized using Synopsys Design Compiler, and fabricated in 0.6um CMOS technology. It supports parallel execution of an adaptive GA, exchanging data with three neighboring GAA chips. Post

layout simulation of the chip showed that it can run at 50 MHz. The GAA system does not perform fitness function evaluation, which is done externally to the system.

2.5 Previous Work on Software-based Genetic Algorithms

There exist many software libraries of GA components. One among them is the GALib, a C++ library of GA objects developed in MIT CADLab in 1995 [18]. The library includes tools for using any GA to do optimization problems using any type of representation and operators. It supports both simple and steady-state GAs. All chromosome initialization, selection, crossover, mutation and replacement can be customized. The library has been tested on DOS, Windows, Mac and UNIX workstations.

Another important work in the field of software-based GAs is GAUL (Genetic Algorithm Utility Library) [19], an open source programming library designed to assist code development for applications that utilize GAs or evolutionary programming. GAUL was developed in C and includes data structures and functions for handling data to implement serial and parallel evolutionary algorithms.

Another addition to existing software based GAs is the Genetic Algorithm Optimization Toolbox (GAOT) [20], developed in Matlab. This toolbox is widely used and is thus a good base for building our hardware library. The toolbox contains a library of GA modules including roulette wheel, tournament selection and normalized geometric selection for the selection process; one-point, arithmetic and heuristic methods for crossover; and various mutation methods. It supports binary and floating point

representations. Our thesis does not involve floating point representations, which are discussed in the Future Work section of Chapter 6.

2.6 Previous Work on Developing this Library

A part of this work involves extending a library of GA modules described in [10]. The library was based on the widely used GAOT Matlab library [34]. The different modules of the GA system, i.e., selection module (roulette wheel selection), crossover (one-point, two-point, arithmetic, uniform) and mutation (flip-bit) were described in VHDL. In addition, random number generators based on linear feedback shift registers (LFSR) and cellular automata (CA) were included. These models were also simulated to check for functionality and performance. The work also involved incorporating the modules into a hardware architecture exploiting the features of pipelining and parallelization.

The motivation behind this thesis is to extend the library and thus have a comprehensive collection of modules that could be employed by other researchers in efficient architectures for specific problems. The HDL simulation results for the GA architecture for several sample problems have been provided here and they have also been compared with their software counterparts.

Chapter 3

Genetic Algorithm Modules and Architecture in Hardware

This chapter describes the various GA modules that have been implemented as a part of this library and the architecture into which the modules have been incorporated.

A part of the library has already been developed [10] as discussed in Chapter 2. Here is a quick review of the modules that were implemented in [10]:

- Random Number Generator: Linear Feedback Shift Register (LFSR) and Multiple LFSR
- Selection Module: Roulette Wheel Selection
- Crossover Module: One-point, Two-point, Uniform and Arithmetic Crossover
- Mutation Module: Flip-bit Mutation

A part of this thesis involves implementing additional modules, thus making the library a comprehensive one. The modules implemented include:

- Random Number Generator: Cellular Automata (CA), Multiple CA, Hybrid CA
- Selection Module: Tournament Selection, Rank Selection, Truncation Selection
- Reproduction Module (combination of crossover and mutation): Half Sibling and a Clone

The modules have been incorporated into a sample architecture for functionality and performance evaluation.

3.1 Random Number Generators (RNGs)

Based on the analysis of RNGs and the results obtained by Martin [21], discussed in [10], LFSR, multiple LFSR, cellular automata (CA) and multiple CA have been included in the library. LFSR based RNG has already been implemented as a part of [10]. This thesis involves implementing CA based RNGs.

3.1.1 Cellular Automata (CA)

A popular RNG for hardware implementation is based on cellular automata (CA). In simple terms, a 1-dimensional cellular automaton is a string of cells with each cell having 2 neighbors, on its left (West) and right (East). At any time step, the value of a cell depends on its previous value and/or its left and right neighbor cell values, based on a set of rules, and the entire string forms the random number. For this implementation Rule 30 is used, according to which, the value of a cell C at time t+1 is given by,

$$C_{(t+1)} = (\text{West}_{(t)} \text{ XOR } (C_{(t)} \text{ OR } \text{East}_{(t)}))$$

Current Pattern	111	110	101	100	011	010	001	000
New State of Center Cell	0	0	0	1	1	1	1	0

Table 3.1 State Transition according to Rule 30

According to Rule 30, as shown in Table 3.1, if the center bit denotes the cell, then the bit to its left is its West cell and the bit to its right is its East cell. And the value of the cell at the next instant (which depends on the current value of the cell and its two neighbors) is provided below each cell. Rule 30 is named so because 30 is the decimal equivalent of 00011110_2 , the rule set which governs the next state of the automaton.

A CA may possess null boundary condition (Figure 3.1(a), where the first and last cells assume their West and East neighbor respectively to be 0) or cyclic boundary condition (Figure 3.1(b), where the CA forms a ring and the West neighbor of the first cell is the last cell and the East neighbor of the last cell is the first cell).

Figure 3.1(a) CA with Null Boundary Condition

Figure 3.1(b) CA with Cyclic Boundary Condition

The result of running this PRNG using a Serial test described by Knuth [23] and analyzed by Martin [21] is shown in Figure 3.2, which is a plot of PRNG values at instant t and $t+1$.

Figure 3.2 Serial Test Result for 1D CA RNG [21]

As can be seen, there is a distinct pattern to numbers as in the case of a LFSR [21].

3.1.2 Multiple Cellular Automata

For better performance, several one dimensional CAs are combined together to form the output. In this implementation, 32 8-bit CAs have been employed and the first bit from each CA has been taken to form a 32-bit random number. Each CA is initialized with different seeds. The output patterns have been run against the Serial Test described by Knuth [23] and the results provided by Martin [21] are shown below,

Figure 3.3 Serial Test Result for 32 CA RNG [21]

3.1.3 Hybrid Cellular Automata

A cellular automaton which yields maximum length binary sequences by making use of a combination of rules is known as a *hybrid CA*. It has been found [24] that a combination of Rule 90,

$$C_{(i)}(t+1) = C_{(i-1)}(t) \text{ XOR } C_{(i+1)}(t)$$

and Rule 150,

$$C_{(i)}(t+1) = C_{(i-1)}(t) \text{ XOR } C_{(i)}(t) \text{ XOR } C_{(i+1)}(t)$$

at appropriate sites can produce maximum length cycles for almost all lengths. The state diagram of a Hybrid CA is shown in Figure 3.4.

Figure 3.4 State Diagram of a 498 Site Hybrid CA for 800 Time Steps [24]

As can be seen, unlike Rule 30, a hybrid CA shows symmetry on the left and right side of the figure. Thus the binary sequences produced by a Hybrid CA fail some random tests because of distribution problems. The fundamental problem with this type of CA is the

generation of sequences with more zeros than ones in one stage and vice versa at another stage. A site spacing of 1 overcomes this problem of spatial correlation. A site spacing of 1 denotes a CA of length $2N$ where outputs are drawn out at alternative sites (cells) to produce a sequence of length N .

3.1.4 Speed-Area Comparison between LFSR and CA

In VLSI implementations, a signal path is modeled as a network of resistors and capacitors as shown in Figure 3.5.

Figure 3.5 RC Representation of a Signal [25]

The delay time of the signal is given as,

$$\tau = (r * c * l^2) / 2$$

where r and c are resistance and capacitance per unit length and l is the length of the signal path. The delay equation shows that the length of the signal path becomes the primary concern in VLSI implementation.

In contrast to LFSR, CA has the property of locality of signal path (without any global signal routing), thus giving CA an advantage in speed over LFSR in VLSI implementations. Figures 3.6 and 3.7 show the contrast in structure between a LFSR and a CA, where D_i is a D flip-flop, CL is combinational logic, and a_i is logical 0 or 1.

Figure 3.6 Structure of a LFSR [25]

Figure 3.7 Structure of a CA [25]

The performance metrics of the hardware implementation of LFSR and CA were studied by Jason et al. [26], where the hardware implementations were simulated using Xilinx Virtex XCV800 FPGA series. The results are shown in Table 3.2

Performance Metrics	1D CA (Rule 30)	1D Hybrid CA (Rule 90-150)	LFSR
Gate Count	700	700	806
Min Timing Delay (ns)	6.429	7.564	17.855
Maximum Frequency	155.54	132.204	56.007

Table 3.2 Various Performance Metrics for LFSR and CA [26]

3.2 Selection Module

Selection is a necessary operator in evolutionary algorithms. Depending on the fitness or objective function values, selection decides on the reproduction of individuals. Selection gives a direction to evolution and guides the search process. The right amount of pressure has to be applied so that good solutions survive and weak solutions die; with too much pressure, the solution converges to local optima, and with too little pressure, there is no progress towards the optimal solution. Hence it becomes a balancing act to choose the right selection operator for the problem at hand.

Roulette wheel selection module was implemented as a part of [10]. And as a part of this work, we have added the following selection operators:

- Tournament selector
- Rank selector
- Truncation selector

3.2.1 Tournament Selection

In this scheme, some number t ($t > 1$) of chromosomes is selected at random from the population and the chromosome with highest fitness gets selected to reproduce. Often tournaments are held only between two individuals (binary tournament) but this method can be generalized to an arbitrary size t called Tournament Size. For example, consider the population in Table 3.3:

Individual	Score
A	4
B	10
C	14
D	7
E	9
F	6

Table 3.3 Example Population for Tournament Selection

In this example, A has 1/36 chance of being selected (chance of A being both the competitors) while the selection probability for the most fit individual C is 11/36. These values do not vary with the score but rather are dependent on the rank of the individual and hence are constant for a particular rank. Thus it can be said that Tournament Selection does not care about the spread of the scores, but only the ranking. The n th ranked individual in a population of size t will have $(2t - 2n + 1) / t^2$ chance of being selected. This puts an upper and lower bound on the chances of an individual being selected for reproduction.

Tournament Selection can also be combined with roulette wheel selection, where the two candidates are selected with roulette wheel probabilities. This is called stochastic tournament selection and its effectiveness is still in question. The performance of a Stochastic tournament selection with its additional call overheads must be evaluated

against a straight tournament selection for the problem at hand. This thesis involves implementation of the straight binary tournament selection.

3.2.2 Rank Selection

The roulette wheel method of selection will have problems when the fitness values differ greatly. For example, if the most fit solution occupies 90% of the wheel, then the probability of any other solution being selected becomes very small. This is one important drawback of fitness proportionate selection schemes.

In a rank-based scheme with N individuals, each individual is assigned a rank based on its fitness, the less fit one being rank 1 and the most fit individual being rank N and selection is based on this ranking rather than on the absolute difference in fitness. Thus it prevents very fit individuals from gaining dominance very early at the expense of less fit ones, which would reduce the population's genetic diversity and hinder attempts to find an acceptable solution. The difference between the two schemes can be clearly seen with the help of Table 3.4 and Figure 3.8 and 3.9.

In Table 3.4, cumulative fitness of chromosome i is the sum of the cumulative fitness value of chromosome $(i-1)$ and the fitness value of chromosome i . The ranking is done based on the fitness value of each chromosome.

Individual	Chromosome	Fitness	Cumulative Fitness	Range of Fitness	Rank
Chromosome1	10010	2	2	0 – 2	1
Chromosome 2	11000	5	7	2 – 7	4
Chromosome 3	00100	3	10	7 – 10	2
Chromosome 4	00001	4	14	10 – 14	3
Chromosome 5	11110	100	114	14 – 114	6
Chromosome 6	100011	6	120	114 - 120	5

Table 3.4 Fitness and Associated Rank for a Sample Population

Figure 3.8 Chart Showing Probability of Selection of Members According to Roulette Selection

Figure 3.9 Chart Showing Probability of Selection of Members According to Rank
Selection

3.2.3 Truncation Selection

Truncation selection is also a selection method used in GAs to select potential candidates for recombination. In truncation selection, the candidate solutions are ranked by fitness and some proportion of the fittest individuals are selected for reproduction. For example, consider the above population and a truncation factor of 0.75 ($3/4^{\text{th}}$ of the fittest individuals are considered for selection). For this example, the roulette wheel is shown in Figure 3.10.

Figure 3.10 Chart Showing Probability of Selection of Members According to Truncation Selection

The pie-chart (Figure 3.10) shows how the less fit solutions are completely eliminated from the selection procedure and the resulting impact on the remaining candidate solutions. The truncation factor cannot be made very low, as it will result in loss of diversity.

3.3 Reproduction Module

The reproduction process involves both crossover and mutation and is responsible for the evolution of the population over generations. For each new solution to be produced, a pair of parents is selected from the mating pool to undergo reproduction. The parents undergo crossover and mutation (this combination is called reproduction) to produce offspring. Some of the most common crossover operators, which were implemented in [10], include: one-point, two-point, uniform and arithmetic crossover.

The new child solution produced typically shares some characteristics of its parents. This process of parent selection, crossover and mutation is repeated until an entire new generation is formed. The new generation thus produced is different from its previous generation. Generally the average fitness of the new set of solutions is higher than its previous generation, since the more fit organisms from the previous generation tend to be selected for breeding.

3.3.1 Half Sibling and a Clone (HSAC)

In HSAC [22] two parents and a random chromosome of the same size are selected. One of the parents (say P1) and the random chromosome act as parents and the other parent (say P2) acts as the decider. For each bit position, if the decider bit is 1 the new child bit will be that of the parent; otherwise the random chromosome bit will be used. This is done for all the bits of the chromosome. This method can be clearly understood from the example in Figure 3.11.

Figure 3.11 Block Diagram Representation of HSAC [22]

3.4 Summary of GA Library Modules

Table 3.5 summarizes the modules which were previously included in the library and the modules that have been newly added as a part of this work.

Module Type	Module Name	Newly Added in this Work
Pseudo-Random Number Generator (PRNG)	1. Linear Feedback Shift Register (LFSR) 2. Multiple LFSR 3. Cellular Automata (CA) 4. Multiple CA 5. Hybrid CA	✓ ✓ ✓
Selection Module	1. Roulette Wheel Selector 2. Tournament Selector 3. Rank Selector 4. Truncation Selector	✓ ✓ ✓
Crossover Module	1. One-Point 2. Two-Point 3. Uniform 4. Arithmetic 5. Half Sibling & Clone	✓
Mutation	1. Flip-Bit	

Table 3.5 Summary of the Existing and Newly Added Modules in the GA Library

3.5 Sample Architecture

This section explains the sample architecture that has been developed to test the modules and verify the functionality and performance. It also discusses some architectures that have been developed by other researchers and the manner in which the modules interact with one another.

Our architecture contains the five essential GA modules, namely, PRNG, selection module, crossover module, mutation module and the fitness evaluation module along with three sets of general purpose registers, one for storing the current population, one for the population fitness, and the last to hold the next generation population members. These various modules are connected together as in Figure 3.12. The way in which the modules interact is described below.

GENETIC ALGORITHM ARCHITECTURE

Figure 3.12 Block Diagram of the Architecture Developed

The PRNG creates the initial population which is stored in *RAM1 for population members* and their fitness, evaluated by the *fitness evaluation block1*, is stored in *RAM2 for fitness values*. The selection module then accesses RAM1 and selects two parents for reproduction. The selected parents undergo crossover and mutation and the offspring are stored in a temporary *RAM3 for new population members*.

The whole process of selection and reproduction is pipelined so that, by the time the two parents undergo crossover and mutation, the next set of parents is already selected and waiting. This helps in increasing the overall speed of the system. Two sets of RAM, one for the old and another for the new population, are necessary to avoid overlap in generations. Otherwise, a new offspring created during an evolution might be employed for further reproduction within the same generation.

The next generation, when ready, is evaluated by *fitness evaluation block2* and both the population members and their fitness values are transferred back to the original RAM. Two sets of fitness evaluation blocks are necessary to allow parallelization of the entire process. The architecture developed is based on the simple GA technique, where an entire population is replaced.

A number of architectures developed by various other researchers have already been discussed as a part of [10]. The modules developed as a part of this work can be incorporated into any one of those architectures. So this section essentially deals only with architectures that have been developed after [10].

In 2005 Nadia et al [15] developed a hardware-only implementation of a genetic algorithm which is problem independent. All the modules in a GA are generic except fitness evaluation, which is problem specific. Thus most work on GA involves a hardware / software co-design. This work designed a neural network for computing the fitness of individual chromosomes, thus making a hardware-only architecture. The overall architecture is shown in Figure 3.13.

Figure 3.13 Overall Architecture of the Hardware Genetic Algorithm(PHGA) [15]

The neural net module has been described in detail in [35]. The neural network makes use of stochastic signal and hence is supposed to reduce the hardware area significantly. The network topology is fully-connected feed forward. A feedforward neural network is an artificial neural network in which the units do not form a closed loop and information flows only in one direction, from the input node to the output node through the hidden nodes. The number of neurons in this implementation is equal to the size of the

individual. And the neural network is supposed to be trained prior to the first use of the GA inside the hardware architecture. A simple feedforward network employed in the design is provided in Figure 3.14

Figure 3.14 Neural Network Architecture [15]

The architecture was compared with hardware engine (HEGA) developed by Scott et al. [12]. The results provided in Table 3.6 clearly show that the architecture is twice as fast as the architecture in [12], even though it occupies five times more area. The tremendous speedup was attributed to the results of massive parallelism employed leading to larger area requirement.

Implementation	Time	Area	Area * Time
HEGA	972	870	845640
PHGA	189	1884	356076

Table 3.6 Comparison of Performance Results between HEGA and PHGA [15]

Chapter 4

DNA Sequence Alignment

4.1 Introduction

In bioinformatics, sequence alignment is a way of comparing the primary sequences of DNA, RNA or proteins in such a way as to identify the regions of similarity. Two sequences may be similar because of divergence from the same ancestor (also called homologous), or because of functional similarity, even though they arose independently (also called analogous). Sequence alignment is used in identifying the conserved regions across a set of sequences hypothesized to be evolutionarily related.

The aligned sequences of nucleotides (in case of DNA) or amino acids (in case of proteins) are represented as rows in a matrix. Gaps are inserted into the alignment so that residues (nucleotides or amino acids) with similar characters are aligned under the same column. An example of two aligned DNA sequences is shown below.

```
CTGCGGG---GGTAAT
  |||||      |||
--GCGG-AGAGG-AA-
```

The ‘-’ represents a gap which is a consecutive run of (usually 20% of sequence length) spaces in the string. If two sequences in an alignment share a common ancestor, mismatches can be interpreted as *point mutations* and gaps as *indels* (insertion or deletion mutations) introduced in one or both lineages after they diverged from one another. While point mutations are often caused by chemicals or error in DNA replication, insertions and deletions are due to transposable elements (sequences of DNA that can

move to different positions inside a cell). The various types of mutations are shown in Figure 4.1.

Figure 4.1 Types of Mutations [27]

4.2 Motivation

Most Multiple Sequence Alignment (MSA) programs use heuristic methods rather than global optimization techniques because to identify alignment between a few sequences of moderate length is itself computationally intensive. And the most widely used heuristic alignment technique is progressive alignment.

Progressive alignment works by aligning two closely related sequences first and then successively aligning the next sequence in the query with the alignment initially produced until the MSA is produced. The initial “most related” pair can be obtained by a simple heuristic search. In this way progressive techniques build up MSA by constructing a phylogenetic tree. The most widely used tool that employs progressive alignment techniques is the Clustal [39] family to which a large number of web portals including GenomeNet [36], EBI [37], EMBNet [38] provide access.

Unfortunately progressive alignment programs are computationally complex, for example, the alignment of a few hundred sequences using ClustalW takes several hours. Owing to the rapid advancements in genome projects biologists would like to compute MSA of increasing sizes in reasonable time.

Hence a suitable approach would be to implement the sequence alignment problem in hardware. The capability to exploit parallelism and incorporate pipelining concepts can produce speedup of several magnitudes. This motivation provided the basis for choosing this project as a test application for applying our GA library.

4.3 GA and Sequence Alignment

It is standard practice to use dynamic programming (DP) for MSA. To align a pair of sequences of length m and n , the time and space complexity are $O(m*n)$. And to perform MSA on m sequences of length n the complexity becomes $O(m^n)$ [28]. This led to the widespread use of progressive techniques for MSA of large sequences. But the drawback in progressive alignment is that it is based on an estimated phylogenetic tree. The whole solution is highly dependent on the initial alignment, which is usually formed without the knowledge of any available data, leading to “local minima” problems.

Stochastic heuristic based approaches like genetic algorithms, simulated annealing, etc., have also been applied to solve the sequence alignment problem. There are two main advantages of using the stochastic techniques [5]. First, there is no strong restriction on the number of sequences to be aligned or the length of the sequence. Second, these techniques can be applied to optimize any function. Also, the implicit parallelism in a GA helps to achieve speedup and better solutions.

4.4 Previous Work

A lot of research work has been done to solve MSA by applying heuristic techniques. This section presents a brief overview of various efforts to combine GA and MSA in software and hardware.

In 2004, W-L Shu et al. [28] developed hardware for a GA to solve the Travelling Salesman Problem (TSP) and MSA. Their approach aimed at solving MSA by just

applying TSP without constructing the phylogenetic tree. In other words, instead of constructing the evolutionary tree, a circular tour of the tree is computed by applying a Traveling Salesman Algorithm. The main advantage of this method is that the scoring can be done with reference to the phylogenetic tree without knowing the structure of the tree. The algorithmic details are not explained here and we have described only the hardware aspects of the architecture. The complete algorithm is explained in [29].

The entire GA system has been partitioned into software and hardware blocks with the software part composed of initial population generation, selection and termination and the hardware block composed of crossover, mutation and fitness evaluation. The hardware architecture is shown in Figure 4.2.

Figure 4.2 Block Diagram of GA Architecture in Hardware for MSA [28]

The major components in the architecture are crossover, inverse mutation, fitness evaluation, range adjustable random number generation and the overall system controller. The detailed description of each of the blocks and its implementation can be obtained from [28]. The hardware modules have been verified and simulated using VHDL. The system was found to have a speedup of 27 for worst case and 44 for best case over its software counterpart [14].

In 2004, Faizal et al. [30] developed a hybrid algorithm which is a combination of a GA and simulated annealing. The GA tries to find new regions of feasible solutions while simulated annealing serves to improvise the solution and prevent convergence to local minima. Anbarasu et al [31] in 2004 implemented a parallel GA (PGA) for sequence alignment. The work involved splitting the entire population into sub-populations which evolve on different processors. Each processor executes a GA sequentially on its sub-population and exchange between the sub-populations is performed with the help of migration operators. The results obtained were found to be better than ClustalW as shown in Table 4.1 (scoring scheme in which a smaller score reflects better alignment).

Test Case	Nseq	Length	CLUSTAL W		PGA	
			Score	CPU Time (in secs)	Score	CPU Time (in secs)
Dfr	4	186	21316	5.440	21104	110
Gcr	8	48	39103	4.790	38903	636
Globin	15	169	4248160	13.019	4222846	1178
S protease	15	292	26285660	22.351	25970235	2815

Table 4.1 Comparison of PGA with ClustalW [31]

4.5 Application of our GA Library to Sequence Alignment

A simple GA starts with a population of initial solutions (represented as bit strings of '1' and '0' called chromosomes). The genetic operators such as crossover, mutation are applied to the population to create a new generation of chromosomes. This process is repeated until the termination condition is met.

The important thing in any application of GA is the representation of the candidate solutions. For this particular application, we have chosen a bit string representation of the solutions. All the sequences are represented as a single bit string with a 1 representing the character and a 0 representing a gap. The standard GA operators are applied to a population encoded into bit strings. The only constraint here is that the total number of 1's for a particular sequence should be equal to the actual length of the sequence. The following example shows how a sample sequence is represented,

Actual Sequence 1: ATGCTATGCG

Sequence after Alignment: AT_G_CTA_TG_CG

Representation: 11010111011011

Actual Sequence 2: GTGATCTACC

Sequence after Alignment: _GTG_ATC_TA_CC

Representation: 01110111011011

Overall String = Rep1 + Rep2 = 1101011101101101110111011011

The fitness function here evaluates the number of matching columns. A match is assigned a score of 1 and a gap or mismatch is assigned a score of 0. The score evaluation of a sample alignment is shown below for reference,

<i>Sequence 1:</i> AT_G_CTA_TG_CG	Gap Length = 4
<i>Sequence 2:</i> T T_G__TA_CG_C_	Gap Length = 6
<i>Score:</i> 6	

This is a very simple representation and any other penalty can be given to gaps and mismatches. Usually leading and ending gaps are given higher penalty when compared to extending gaps. The usually allowed total gap length is 20% of the length of the pattern and the same rule is applied in our system also. Gap length can be defined as the actual number of single character gaps (in the above example that illustrates scoring, Sequence 1 contains 4 gaps and Sequence 2 contains 6 gaps).

The architecture has been designed to perform pair-wise DNA sequence alignment but can be extended to multiple sequence alignment also. The system has been used to compare 5 sets of sequences each of length 10, 20, 30, 40 and 50 nucleotides (building blocks of DNA) and the results are compared with ClustalW. ClustalW employs progressive alignment techniques (explained in Section 4.2) to perform sequence alignment and it is not based on GA. The discussion of the results and the area/time report is presented in the next chapter.

Chapter 5

Performance Analysis

In this thesis work, several GA modules have been implemented in VHDL, in addition to the modules that were already developed as a part of [10]. A complete list of modules implemented in hardware is given in Table 3.4. The fitness module is designed according to the GA application. Here, the GA system is applied to two maximization functions. The different modules are incorporated into a sample architecture shown in Figure 5.1 and the performance of each combination is analyzed by comparing the results obtained in each case.

The developed library is also applied to DNA sequences to perform pairwise alignment. DNA sequences of length 50, 40, 30, 20 and 10 nucleotides are aligned and the results are compared with a software program called ClustalW developed by European Bioinformatics Institute (EBI) [37]. The rest of this chapter presents the results obtained and the speedup achieved when compared to the software counterpart.

5.1 Experimental Setup

The GA system consists of a Random Number Generator (RNG), selection module, crossover module, mutation module and fitness function evaluation module. These modules are incorporated into the sample architecture shown in Figure 5.1.

GENETIC ALGORITHM ARCHITECTURE

Figure 5.1 Architecture of the GA System Implemented in this Work

The PRNG is used to produce the initial population and also to generate random numbers for the crossover, selection and mutation modules. The fitness of each individual chromosome is determined by the fitness evaluation module and each chromosome along with its fitness value is stored in a RAM. Once fitness evaluation is completed, the selection module selects two parents for reproduction (crossover and mutation). The selected parents undergo crossover and mutation to produce offspring. The offspring thus produced along with the fitness values replace the old generation in the RAM. This process is repeated until the maximum number of generations (termination condition) is reached.

The different types of modules are incorporated into the GA system and the performance of each combination is evaluated. The modules have been implemented in VHDL and simulations were carried out using ModelSim SE 6.1 simulator. The models were simulated on a UNIX platform on SUN workstations.

5.2 Functional Verification

The GA modules were functionally verified in two steps. First, the modules were individually verified to operate correctly under all possible conditions and inputs. Second, the modules were incorporated into the architecture illustrated in Figure 5.1 and the overall system was applied to maximize two functions as in [10]:

- $f(x) = 2x$
- $f(x) = 2x^3 - 45x^2 + 100x$

The maximum value obtained by the GA system is a measure of the performance of the system. And the successful completion of these two steps ensures the proper functioning of the individual modules and the architecture.

For all simulations, a population size of 8 was used and the GAs were simulated for 16 runs, each with a different seed to the PRNG. The fitness values are 12 bits in length. The VHDL simulations verified the correct functionality of all the modules and the architecture.

5.3 Performance Analysis

Once the functionality of all the modules was verified, the next step was to perform performance analyses. The GA system was used to maximize the two functions mentioned above and the maximum value obtained by a GA determined the performance of the system. A summary of the conclusions drawn from [10] is provided below:

- Initial population generated by multi-LFSR is more suited for our GA than a simple LFSR based PRNG.
- The type of crossover operator employed did not have a major impact on the solution except for uniform crossover, which did not perform as expected.

The first step in our performance analyses involves varying the crossover operator keeping all other modules constant. A LFSR was used to generate the initial population. To maximize $f(x) = 2x$, a GA system with the following five combinations was employed:

- LFSR based RNG with roulette selection and one-point crossover
- LFSR based RNG with roulette selection and two-point crossover
- LFSR based RNG with roulette selection and uniform crossover
- LFSR based RNG with roulette selection and arithmetic crossover
- LFSR based RNG with roulette selection and half-sibling and clone crossover

The first four comparisons were already done as a part of [10] and the following Table 5.1 compares the performance of half-sibling and clone crossover with the already existing crossover operators.

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval (95%)
1	LFSR, Roulette, One-point	426	510	474.125	29.62403	(459.61,488.64)
2	LFSR, Roulette, Two-point	472	508	497	13.77921	(490.25,503.75)
3	LFSR, Roulette, Uniform	402	496	443.25	23.61779	(431.68,454.83)
4	LFSR, Roulette, Arithmetic	438	502	492.375	17.31618	(483.89,500.96)
5	LFSR, Roulette, Half – Sibling Clone	494	510	505.125	6.810531	(501.79,508.46)

Table 5.1 Performance Results with Various Crossover Operators for Function f(x)

$$= 2x$$

The functions were maximized over the range $0 \leq x \leq 255$, hence the maximum possible value is 510. The tabulated results clearly show that a GA system with half-sibling and clone as crossover operator produces superior results when compared to the other crossover operators. This could be due to the fact that half-sibling and clone operator involves more randomness in its offspring generation process and thus is able to cover a wider area in the sample space.

The next set of experiments was performed by changing the PRNG from LFSR to Multi-LFSR. The results obtained for the function $f(x) = 2x$ with different GA systems is provided in Table 5.2.

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval (95%)
1	MultiLFSR, Roulette, 1Pt	438	510	479.6	37.18602	(447.01,512.19)
2	MultiLFSR, Roulette, 2Pt	498	510	503.6	4.560702	(499.60,507.6)
3	MultiLFSR, Roulette, Uniform	440	510	485.2	29.88645	(459.00,511.4)
4	MultiLFSR, Roulette, Arithmetic	474	510	485.2	14.25482	(472.71,497.7)
5	MultiLFSR, Roulette, HalfSiblingClone	506	510	509.5	1.322876	(508.83,510.17)

Table 5.2 Performance Results with Various Crossover Operators for Function

$$f(x)=2x$$

A comparison of simulation results for LFSR based GA and Multi-LFSR based GA is shown in Figure 5.2. The results show that all the GA systems attained the maximum possible value of 510.

Figure 5.2 Comparison of Results Between LFSR and Multi-LFSR Based GA Implementations for Function $f(x)=2x$

It can be seen that the average for all the different GA systems has increased with multi-LFSR. However, arithmetic crossover did not perform as expected in the previous implementation (Table 5.1). The remaining combinations show considerable improvement, and as before, half-sibling and clone shows superior results over the others.

The same combination of operators were also applied to the second function, $f(x) = 2x^3 - 45x^2 + 100x$ to be maximized. For a LFSR based GA system five possible combinations

exist that are similar to the implementations for the first fitness functions. The results obtained are provided in Table 5.3. As mentioned before, the first four comparisons were already performed as a part of [10] and this work involves comparing the half-sibling and clone operator with the remaining crossover operators.

S No	Combination	Min Value	Max Value	Average Value	Standard Deviation	Confidence Interval (95%)
1	LFSR, Roulette, 1Pt	18120672	29947108	24175125	5369298	(25240797, 34653419)
2	LFSR, Roulette, 2Pt	20288128	29947108	26011529	3902130	(26526803, 33367413)
3	LFSR, Roulette, Uniform	21433824	27124452	23617961	2639683	(24810710, 29438194)
4	LFSR, Roulette, Arithmetic	28866757	28866757	28866757	0	(28866757, 28866757)
5	LFSR, Roulette, HalfSiblingClone	30313125	30313125	30313125	0	(30313125, 30313125)

Table 5.3 Performance Results with Various Crossover Operators for Function $f(x)$

$$= 2x^3 - 45x^2 + 100x$$

As can be seen, only a GA system with half-sibling and clone was able to attain the maximum possible value. None of the other combinations was able to reach the maximum value even once.

As before, Multi-LFSR based GA system was applied to the second maximization function $f(x) = 2x^3 - 45x^2 + 100x$ and the results are provided in Table 5.4

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval(95%)
1	MultiLFSR, Roulette, 1Pt	28512500	30313125	29301249	781139.9	(28616563, 29985936)
2	MultiLFSR, Roulette, 2Pt	28161153	30313125	29447642	936644.3	(28626652, 30268632)
3	MultiLFSR, Roulette, Uniform	18382721	30313125	25964649	4674096	(21867698, 30061600)
4	MultiLFSR, Roulette, Arithmetic	28866757	28866757	28866757	0	(28866757, 28866757)
5	MultiLFSR, Roulette, HalfSiblingClone	30313125	30313125	30313125	0	(30313125, 30313125)

Table 5.4 Performance Results with Various Crossover Operators for Function $f(x)$

$$= 2x^3 - 45x^2 + 100x$$

Figure 5.3 below illustrates the comparison between the different GA implementations used to maximize the fitness function $f(x) = 2x^3 - 45x^2 + 100x$. It can be observed that the results are similar to those obtained for $f(x)=2x$ (four out of the five possible combinations were able to reach the maximum possible value). The architecture takes 10 clock cycles to complete an iteration, which comprises of selection, crossover and mutation operations.

Figure 5.3 Comparison of Results Between LFSR and Multi-LFSR Based GA

Implementations for Function $f(x) = 2x^3 - 45x^2 + 100x$

From the four tables (5.1, 5.2, 5.3, 5.4), we can see that half-sibling and clone has reached the maximum possible value in all the cases and thus has out-performed all other crossover operators. As can be observed, half-sibling and clone operator has reached the maximum possible value when employed with both LFSR and multi-LFSR. Another important observation to be made is that GA systems that employ MultiLFSR have shown better results when compared to those which employ LFSR.

The previous comparisons did not include cellular automata (CA) which is another form of a PRNG, developed as a part of this thesis. Hence the next few experiments involve

varying the PRNG and keeping the other modules the same. Half-sibling and clone, which performed better than the other crossover operators, was used for crossover and roulette wheel was used as selection operator. The various possible combinations of GA systems analyzed for performance are:

- LFSR with roulette selection and half-sibling and clone crossover
- Multi-LFSR with roulette selection and half-sibling and clone crossover
- Multi-CA with roulette selection and half-sibling and clone crossover
- Hybrid CA with roulette selection and half-sibling and clone crossover

These various GA systems were applied to the function $f(x) = 2x$ and the results are presented in Table 5.5 and plotted in Figure 5.4

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval (95%)
1	LFSR, Roulette, HalfSiblingClone	494	510	505.125	6.594268	(498.53,511.72)
2	MultiLFSR, Roulette, HalfSiblingClone	506	510	509.5	1.322876	(508.85,510.15)
3	MultiCA, Roulette, HalfSiblingClone	494	510	507	6.244998	(503.94,510.06)
4	HybridCA, Roulette, HalfSiblingClone	506	510	509.5	1.322876	(508.85,510.15)

Table 5.5 Performance Results with Various PRNGs for Function $f(x)=2x$

Figure 5.4 Results with Various PRNGs for Function $f(x)=2x$

The same combination of GA systems was applied to the second maximization function

$f(x) = 2x^3 - 45x^2 + 100x$ and the results are presented in Table 5.6

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval (95%)
1	LFSR, Roulette, HalfSibling Clone	30313125	30313125	30313125	0	(30313125,30313125)
2	MultiLFSR, Roulette, HalfSibling Clone	30313125	30313125	30313125	0	(30313125,30313125)
3	MultiCA, Roulette, HalfSibling Clone	30313125	30313125	30313125	0	(30313125,30313125)
4	HybridCA, Roulette, HalfSibling Clone	30313125	30313125	30313125	0	(30313125,30313125)

Table 5.6 Performance Results with Various PRNGs - Function $f(x)=2x^3-45x^2+100x$

Table 5.6 shows that all the GA systems with various PRNGs perform equally well and have attained the maximum possible value on all 16 runs, and hence do not provide any conclusive result. However, on analyzing Table 5.5 we can infer that GA systems with multi-LFSR and hybrid CA show better results when compared with the remaining combinations. The reason for this superior performance could be because of the fact that both multi-LFSR and hybrid CA produce better quality random numbers as discussed in [10].

The last set of experiments involved varying the selection operator while keeping the remaining modules fixed. From the analysis done so far, it can be inferred that half-sibling and clone shows superior performance when compared to other crossover operators. Similarly, GA systems with multiLFSR and hybridCA have shown higher fitness values when applied to both the maximization functions. Thus to perform the next set of experiments these two operators are kept the same while the selection operator is varied. The possible combinations for the GA system are:

- HybridCA, rank selection, half-sibling and clone
- HybridCA, truncation selection, half-sibling and clone
- HybridCA, tournament selection, half-sibling and clone
- HybridCA, roulette selection, half-sibling and clone

The fitness function $f(x) = 2x$ is maximized by the different GA implementations and the simulated results are presented in Table 5.7. The initial population is generated by hybridCA and each system is simulated for 16 runs with different seeds and the results are tabulated. The results shown in Table 5.7 are plotted in Figure 5.5.

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval (95%)
1	HybridCA, Rank Select, HalfSiblingClone	502	510	509.125	1.99609	(508.15,510.1)
2	HybridCA, Truncation Select, HalfSiblingClone	510	510	510	0	(510,510)
3	HybridCA, Tournament Select, HalfSiblingClone	510	510	510	0	(510,510)
4	HybridCA, Roulette, HalfSiblingClone	506	510	509.5	1.322876	(508.85,510.15)

Table 5.7 Performance Results with Various Selection Operators for Function

$$f(x) = 2x$$

Figure 5.5 Results for Various Selection Operators for Function $f(x) = 2x$ with Hybrid CA and Half-Sibling and Clone

The results show that all the GA implementations were able to reach the maximum value of 510. In particular, GA systems that employed truncation and tournament selector were able to achieve the same maximum value in all 16 runs. The same set of implementations were also applied to maximize the function $f(x) = 2x^3 - 45x^2 + 100x$. The results obtained are shown in Table 5.8 and they corroborate the fact that systems that employ truncation or tournament selection show improved performance.

S No	Combination	Min Value	Max Value	Average	Standard Deviation	Confidence Interval(95%)
1	HybridCA, Rank Select, Half-SiblingClone	28866757	30313125	30176974.875	359107.2	(30001015.58, 30352934.18)
2	HybridCA, Truncation Select, Half-SiblingClone	30313125	30313125	30313125	0	(30313125, 30313125)
3	HybridCA, Tournament Select, Half-SiblingClone	30313125	30313125	30313125	0	(30313125, 30313125)
4	HybridCA, Roulette, Half-SiblingClone	30313125	30313125	30313125	0	(30313125, 30313125)

Table 5.8 Performance Results with Various Selection Operators for Function

$$f(x) = 2x^3 - 45x^2 + 100x$$

The advantage of truncation selection over rank selection is that in truncation selection only a portion of the overall population is deemed fit for reproduction. In other words, the less fit individuals are not considered for selection operation, thus making sure that offspring are produced only from highly fit parents and the less fit parents play no role in the reproduction process. Though this might seem advantageous there is always the possibility of the system getting stuck in local maxima, thus losing the chance of exploring the wide sample space. Or, in other words, truncation selection might lead to loss of diversity if not properly employed.

5.4 Results of DNA Sequence Alignment

The modules from the library are incorporated into architecture and applied to DNA sequence alignment. In the previous section, the first set of experiments showed that half sibling and clone has superior performance when compared to all crossover operators. It can also be seen that the results produced by a GA system employing multi-LFSR and hybrid CA show better performance. The final set of experiments showed that both truncation and tournament selection perform equally well and better than other selection operators.

Taking these results into consideration, we design a GA system with hybrid CA, tournament selection and half sibling and clone as the constituting modules. The system was applied to DNA sequences 50, 40, 30, 20 and 10 nucleotides in length. The simulations were run for 10 different datasets for each length and each dataset was run 10 times with different seeds, constituting a total of 500 experiments. Each simulation was run for 3000 iterations. Mentor Graphics tool ModelSim SE version 6.1 was used for performing simulations. The average, standard deviation and 90% confidence interval (CI) values have been computed for each data set (each case is run 10 times). The results are compared with Cluastalw values and presented in Tables 5.9 – 5.13.

S No	ClustalW	GA Max	GA Min	GA Avg	Standard Deviation	Confidence Interval (90%)
1	21	21	19	19.9	0.831	(19.22, 20.58)
2	26	25	24	24.6	0.490	(24.20, 25.00)
3	22	20	19	19.9	0.300	(19.66, 20.14)
4	21	21	20	20.7	0.458	(20.33, 21.07)
5	21	20	20	20	0	(20, 20)
6	18	21	18	18.9	1.138	(17.98, 19.83)
7	15	15	14	14.7	0.458	(14.33, 15.07)
8	16	16	15	15.8	0.4	(15.47, 16.13)
9	19	19	17	18.6	0.663	(18.06, 19.14)
10	20	20	19	19.7	0.458	(19.33, 20.07)

Table 5.9 Sequence Alignment Results for Patterns of Length 50 (Each Case Repeated 10 Times)

S No	ClustalW	GA Max	GA Min	GA Avg	Standard Deviation	Confidence Interval (90%)
1	17	16	14	15.4	0.8	(14.75, 16.05)
2	16	16	15	15.8	0.4	(15.47, 16.13)
3	14	15	12	13.7	0.781	(13.06, 14.34)
4	16	15	12	13.7	0.781	(13.06, 14.34)
5	13	14	13	13.2	0.4	(12.87, 13.53)
6	12	13	11	12	0.633	(11.48, 12.52)
7	15	14	13	13.5	0.5	(13.09, 13.90)
8	14	14	12	13	0.775	(12.37, 13.63)
9	13	14	13	13.2	0.4	(12.87, 13.53)
10	13	13	12	12.7	0.458	(12.33, 13.07)

Table 5.10 Sequence Alignment Results for Patterns of Length 40 (Each Case Repeated 10 Times)

S No	ClustalW	GA Max	GA Min	GA Avg	Standard Deviation	Confidence Interval (90%)
1	13	11	11	11	0	(11, 11)
2	10	14	12	12.2	0.6	(11.71, 12.69)
3	13	10	10	10	0	(10, 10)
4	11	11	10	10.6	0.490	(10.20, 11.00)
5	9	12	10	10.8	0.6	(10.31, 11.29)
6	11	10	10	10	0	(10, 10)
7	11	11	10	10.7	0.458	(10.33, 11.07)
8	10	10	8	9.4	0.8	(8.75, 10.05)
9	11	11	10	10.6	0.490	(10.20, 11.00)
10	10	10	9	9.6	0.490	(9.20, 10.00)

Table 5.11 Sequence Alignment Results for Patterns of Length 30 (Each Case Repeated 10 Times)

S No	ClustalW	GA Max	GA Min	GA Avg	Standard Deviation	Confidence Interval (90%)
1	11	10	7	9.4	1.02	(8.57, 10.23)
2	8	9	8	8.3	0.458	(7.93, 8.67)
3	8	9	8	8.1	0.3	(7.86, 8.34)
4	5	8	7	7.5	0.5	(7.09, 7.90)
5	6	7	6	6.3	0.458	(5.93, 6.67)
6	5	8	7	7.2	0.4	(6.87, 7.53)
7	6	6	5	5.7	0.458	(5.33, 6.07)
8	7	7	6	6.3	0.458	(5.93, 6.67)
9	5	5	5	5	0	(5,5)
10	5	6	5	5.2	0.4	(4.87,5.53)

Table 5.12 Sequence Alignment Results for Patterns of Length 20 (Each Case Repeated 10 Times)

S No	ClustalW	GA Max	GA Min	GA Avg	Standard Deviation	Confidence Interval (90%)
1	5	5	5	5	0	(5, 5)
2	4	5	4	4.4	0.49	(4.00, 4.80)
3	6	5	4	4.1	0.3	(3.86, 4.34)
4	2	3	2	2.6	0.49	(2.20, 3.00)
5	1	4	2	3	0.447	(2.64, 3.36)
6	3	5	3	3.9	0.539	(3.46, 4.39)
7	2	3	2	2.7	0.458	(2.33, 3.07)
8	3	3	1	2.6	0.663	(2.06, 3.14)
9	3	3	3	3	0	(3,3)
10	3	3	3	3	0	(3,3)

Table 5.13 Sequence Alignment Results for Patterns of Length 10 (Each Case Repeated 10 Times)

The difference between ClustalW value and GA value (GA max, GA Min and GA Average) is calculated for each experiment and its average over the 10 different data sets for each length has also been tabulated. This estimate along with number of confidence intervals that include the ClustalW value has been tabulated and presented in Table 5.14.

S No	Pattern Length	Avg (ClustalW – GA Max)	Avg (ClustalW – GA Min)	Avg (ClustalW – GA Avg)	No. CIs that include ClustalW Value
1	50	0.1	1.4	0.62	6
2	40	-0.1	1.5	0.68	6
3	30	-0.1	0.9	0.41	7
4	20	-0.9	0.2	-0.3	8
5	10	-0.7	0.3	-0.23	9

Table 5.14 Average Difference between ClustalW and GA Values for 10 Sets (Positive Difference Indicates Better Performance by ClustalW)

A positive difference value in the above Table indicates that the ClustalW performs better than our GA. But as can be seen, the difference is very small and the maximum value produced by our GA out of the ten cases is better than ClustalW for four pattern lengths. A plot of the last column, which denotes the number of instances in which the % confidence interval includes the ClustalW value is shown in Figure 5.6.

Figure 5.6 Number of Instances of % Confidence Interval (90%) that Include ClustalW Value

From the plot (Figure 5.6) we can say with 90% confidence that our GA reaches or exceeds the ClustalW values on 6 out of 10 cases in the worst case scenario and 9 out of 10 cases in the best case scenario.

5.5 Area and Timing Report

The entire architecture was implemented on Altera FPGA devices to get area estimates. The Cyclone EPIC6Q240 family and the Stratix II EP2S5F484C3 family of devices were used for testing purposes. The system can align sequences of length 20 when implemented on the Cyclone family and length 50 when implemented on the Stratix II

devices. The actual area estimate in terms of look-up tables (LUTs), for these configurations, is provided in Table 5.15. The maximum pattern length that can be handled by a FPGA device is limited either by its size or the number of pins available.

FPGA Family	Max Length Possible	LUTs Used/ Total LUTs	% LUTs Used	Pins Used / Total Pins	% Pins Used
Cyclone	20	5041 / 5980	84 %	175 / 185	94 %
Stratix	50	9063/ 12480	72%	315 / 343	91%

Table 5.15 Area Report when Implemented on Altera FPGA Devices

Speedup, in our case, can be defined as the ratio of run time of the software (ClustalW) to run time of the hardware (GA). The speedup obtained by our hardware GA over its software counterpart ClustalW is provided in Table 5.15. Our system can function at a clock frequency of 50 MHz. This value is obtained from the report generated as a result of implementing the architecture on the Altera board using Quartus II software. The GA takes 16 clock cycles to load the initial population and 28 clock cycles per iteration to produce the next generation. With a clock period of 20 ns our system can complete 3000 iterations in 1680.38 μ s. The time taken by ClustalW for a particular data set divided by the time taken by GA gives the speedup achieved. The speedup values for the different data sets of varying length are provided in Table 5.16.

Length / Data Set	50	40	30	20	10
1	6012	6012	5994	5994	5988
2	5982	5982	5970	5970	5958
3	6042	6036	6018	6006	5946
4	5976	5970	5970	5964	5958
5	6012	6012	6006	5994	5994
6	6012	6006	6006	6006	6000
7	5964	5952	5952	5946	5940
8	5970	5970	5964	5964	5964
9	5970	5964	5964	5952	5952
10	5964	5958	5958	5946	5940
Average Speedup	5990.4	5986.2	5980.2	5974.2	5964

Table 5.16 Speedup over ClustalW for Different Data-sets of Varying Length

From Table 5.16, it can be inferred that the hardware GA architecture achieves a significant speedup over ClustalW.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

A part of this thesis involved developing an extended library of hardware modules for GAs, and an example architecture which other researchers can customize for specific applications. The various modules of the GA system are described in VHDL. The different modules are incorporated into a sample architecture and applied to various optimization functions. A comparison of the different configurations has also been provided.

The remaining portion of the thesis involved applying the developed architecture to DNA sequence alignment. The results obtained were compared with those of a software counterpart called ClustalW, a multiple sequence alignment program for DNA or proteins, developed by European Bioinformatics Institute (EBI).

Chapter 5 dealt with the performance analyses of the different configurations of a GA and its application to sequence alignment. Here is a summary of the results provided in chapter 5:

- Simulation results clearly show that a GA system with half-sibling and clone as crossover operator produces superior results when compared to other crossover operators.
- The initial population generated also had an impact on the final solution reached by the GA. The multi LFSR and hybrid CA based GA systems have shown to reach the optimal value in more cases when compared to the other configurations.

- Even though all four selection operators have nearly same average values, GA systems employing truncation and tournament selectors were able to reach the maximum possible value for both maximization functions and in all experimental runs.
- The sequence alignment results show that our GA system is functionally correct and produces results comparable those from ClustalW. This indicates that our generic architecture components can provide acceptable results compared to the customized ClustalW approach. Our hardware GA also achieves an average speedup of 5979 when compared to the ClustalW run time.

6.2 Future Work

The modules that were developed deal only with binary representations. This work could be extended to incorporate floating point representations. The floating point arithmetic can be realized in accordance with the IEEE 754 standard.

The architecture and the constituting modules have been applied to DNA sequence alignment. The same architecture can be also extended to be compatible with protein sequences. And, as can be observed, smaller pattern lengths can be handled by a single FPGA. To apply the system to sequences of longer lengths, the design can be implemented on multiple FPGAs. Initial population of random alignments can be split into subsequences and assigned to different processors. Each processor, disjointly and in parallel can execute sequential GA on its population of subsequences. Afterwards, all the subsequences are combined to calculate the overall score. This process can continue with separate evolution of subsequences for certain number of generations.

Bibliography

- [1] Goldberg, D.E., Genetic Algorithms in Search, Optimization and Machine Learning, Addison Wesley Longman, Inc., 2002.
- [2] Schnecke, V. and Vornberger, O., "An adaptive parallel genetic algorithm for VLSI-layout optimization," Proc. 4th Conf. on Parallel Problem Solving from Nature (PPSN IV), 1996.
- [3] De Jong, K.A., Spears, W.M., "Using genetic algorithms to Solve NP-complete problems", Proc. Int'l Conference on Genetic Algorithms and their Applications, 1989.
- [4] <http://www.nd.edu/~gmadey/bio05/ClassNotes/sa.pdf> - Date Accessed 01/28/2008.
- [5] Notredame, C., Higgins, D.G., "SAGA: sequence alignment by genetic algorithm", Nucleic Acids Research, pp 1515-1524, 1996.
- [6] Seijas, J., Morató, C., Andina, D., Vega-Corona, A., "Improving the efficiency of multiple sequence alignment by genetic algorithms", 7th International Work-Conference on Artificial and Natural Neural Networks, 2003.
- [7] Chiang, J., Studniberg, M., Shaw, J., Seto, S., Truong, K., "Hardware accelerator for genomic sequence alignment", Proceedings of the 28th IEEE EMBS Annual International Conference, 2006.
- [8] Beasley, D., Bull, D.R., Martin, R.R., "An Overview of Genetic Algorithms: Part 1 Fundamentals", University Computing, 1993.
- [9] Holland, J.H., "Adaptation in Natural and Artificial Systems", The University of Michigan Press, Ann Arbor, Michigan, 1975.
- [10] Narayanan, S., "Hardware implementation of genetic algorithm modules for intelligent systems", Masters Thesis, University of Cincinnati, 2005.

- [11] Goldberg, D.E., Deb, K., “A comparative analysis of selection schemes used in genetic algorithms”, Foundations of Genetic Algorithms, Morgan Kauffmann, pp 69-93, 1991.
- [12] Scott, S.D., Seth, S., Samal, A., “A hardware engine for genetic algorithms”, Technical Report, University of Nebraska-Lincoln, 1997.
- [13] Nedjah, N., Mourelle, L.M., “Hardware architecture for genetic algorithms”, Innovations in Applied Artificial Intelligence, Springer Berlin / Heidelberg, pp 554-556, 2005.
- [14] Graham, P., Nelson, B., “A hardware genetic algorithm for the traveling salesman problem on SPLASH2”, Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications, 1995.
- [15] Nedjah, N., Mourelle, L.M., “A reconfigurable parallel hardware for genetic algorithms”, Parallel Evolutionary Computations, Springer Berlin / Heidelberg, pp 56-69, 2006.
- [16] Koonar, G., Areibi, S., Moussa, M., “Hardware implementation of genetic algorithm for VLSI design CAD”, Master’s Thesis, University of Guelph, 2003.
- [17] Wakabayashi, S., Koide, T., Toshine, N., Yamane, M., Ueno, H., “Genetic algorithm accelerator GAA II”, Proc. of ASP DAC, IEEE Design Automation Conference, 2000.
- [18] Wall, M., “GALib: A C++ library of genetic algorithm components”, Massachusetts Institute of Technology, <http://lancet.mit.edu/ga/> - Date Accessed 10/05/2007
- [19] <http://gaul.sourceforge.net/> - Date Accessed 11/15/2007
- [20] Houck, C., Joines, J., Kay, M., “A genetic algorithm for function optimization: a Matlab implementation”, NCSU-IE Technical Report, 1995.

- [21] Martin. P., An analyses of random number generators for a hardware implementation of genetic programming using FPGAs and Handel-C, Technical Report, University of Essex, 2002.
- [22] Gantla, D., Abdel-Aty-Zohdy, H.S., Ewing, R.L., “New genetic algorithm Approach for dynamic sensor measurements characterization”, 45th IEEE International Midwest Symposium on Circuits and Systems, 2002.
- [23] Knuth, D.E., Seminumerical Algorithms, Volume 3, Addison-Wesley Publishing Company, 1969.
- [24] Hortensius, P.D., McLeod, R.D., Card, H.C., “Parallel random number generation for VLSI systems using cellular automata”, IEEE Transaction on Computers, Vol 8, 1989.
- [25] Chuanwu, Z., Libin, L., “VLSI Characteristics of cellular automata and LFSR”, IEEE International Symposium on Communications and Information Technology, 2005.
- [26] Isaacs, J.C., Watking, R.K., Foo, S.Y., “Cellular automata: maximal performance and minimal space FPGA implementations”, Engineering Applications of Artificial Intelligence, ScienceDirect, 2003.
- [27] <http://en.wikipedia.org/wiki/Mutation> - Data Accessed 01/10/2008
- [28] Shu, W-L., Wu, C.C., Lai, W-C., “Design the hardware of genetic algorithm for TSP and MSA”, Knowledge-Based Intelligent Information and Engineering Systems, Springer Berlin / Heidelberg Publication, pp 1260-1267, 2004.
- [29] Korostensky, C., Gonnet, G., “Near optimal multiple sequence alignment using travelling salesman approach”, Proceedings of String Processing & Information Retrieval Symposium & International Workshop on Groupware, IEEE Computer Society, 1996.

- [30] Omar, M.F., Salam, R.A., Rashid, N.A., Abdullah, R., “Multiple sequence alignment using genetic algorithm and simulated annealing”, Intl. Conference on Information & Communication Technologies: From Theory to Applications, IEEE, 2004.
- [31] Anbarasu, L.A., Narayanasamy, P., Sundararajan, V., “Multiple sequence alignment using parallel genetic algorithms”, Simulated Evolution and Learning: Second Asia-Pacific Conference on Simulated Evolution and Learning, SEAL '98, Canberra, Australia, November 1998. Selected Papers, 2004.
- [32] Spears, W.M., “Using neural networks and genetic algorithms as heuristics for NP complete problems”, Proceedings of the International Joint Conference on Neural Networks, 1989.
- [33] Darwin, C., The Origin of Species by Means of Natural Selection, or the Preservation of favoured Races in the Struggle for Life, 1859.
- [34] Mazumder, P. and Rudnick, E.M., Genetic algorithms for VLSI designs, layout and test automation, Prentice Hall, 1999.
- [35] Nadjah, N., Mourelle, L.M., “Reconfigurable hardware architecture for compact and efficient stochastic neuron”, Artificial Neural Nets Problem Solving Methods, Springer Berlin / Heidelberg, Volume 2687, pp 17-24, 2003.
- [36] <http://www.genome.jp/> - Date Accessed 01/10/2008.
- [37] <http://www.ebi.ac.uk/> - Date Accessed 01/10/2008.
- [38] <http://www.embnet.org/> - Date Accessed 01/10/2008.
- [39] <http://www.ebi.ac.uk/Tools/clustalw2/index.html> - Date Accessed 01/28/2008.