UNIVERSITY OF CINCINNATI

Date: _____

I, _____,
hereby submit this work as part of the requirements for the degree of:

in:

It is entitled:

This work and its defense approved by:

Chair: _____

Hardware Implementation of Genetic Algorithm

Modules for Intelligent Systems

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

Master of Science

in the Department of Electrical and Computer Engineering

and Computer Science

of the College of Engineering

2005

by

Shruthi Narayanan

B.E., Muffakham Jah College of Engineering and Technology, 2002

Committee Chair: Dr. Carla Purdy

Abstract

A Genetic Algorithm (GA) is an intelligent search strategy supported by operations inspired by biological evolution. A typical GA for a particular application is a robust optimization algorithm that mimics the Darwinian theory of evolution. Although a GA is able to find very good solutions for a variety of applications, it typically requires many computations and iterations to be effective, and the amount of time consumed by these computations and iterations is enormous. Thus, software implementations of GAs applied to increasingly complex problems and large search spaces can cause unacceptable delays. An alternative to this approach is the hardware implementation of GAs in order to achieve tremendous speedup over software counterparts by exploiting the inherent parallelism of the GA paradigm.

This thesis presents the design of libraries of hardware modules for a GA system – one library in the Hardware Description Language (HDL) Verilog HDL and one in the language VHDL. Each library is based on a widely used MATLAB library. The different modules of the GA system, i.e., selection modules (roulette wheel selection), crossover modules (one-point crossover, two-point crossover, arithmetic crossover, uniform crossover), the mutation module, example fitness modules and the random number generators are described in hardware using the HDLs. These Verilog HDL and VHDL RTL models are simulated to check for functionality and performance. The HDL

modules can be used by other researchers to implement GAs in hardware. In this work the modules are also incorporated into an example GA architecture which exploits the features of pipelining and parallelization. These hardware modules can be synthesized to obtain the gate-level netlist and can be further mapped to Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). The HDL simulation results and performance analysis for the GA architectures for several example problems are presented here and these results are compared to similar GAs implemented in software and in problem-specific custom hardware architectures. The modules in the library can be used by researchers to implement efficient architectures for specific problems of interest.

Acknowledgements

I would like to express my gratitude to my advisor Dr. Carla Purdy for her support and motivation throughout this thesis work. I would also like to thank Dr. George Purdy and Dr. Robert Ewing for being on my committee.

A special thanks to my parents for their support and patience that has seen me through my endeavor. I am grateful to my friends for their help and patience throughout my studies.

Lastly, I would like to thank my lab mates Vikram, Madhuri, Easwari, Jothiram, Aravind, Shruti and Rajesh for their constant advice and inspiration.

Table of Contents

1. Introduction	7
1.1 Motivation	8
1.2 Introduction to Genetic Algorithms	9
1.3 Introduction to Verilog HDL and VHDL	10
1.4 Organization of the Thesis	11
2. Overview of Genetic Algorithms	12
2.1 Terminology of Genetic Algorithms	13
2.2 A Simple Genetic Algorithm	14
2.3 The Steady-State Algorithm	17
2.4 Previous Work in Hardware Genetic Algorithms	20
2.5 Genetic Algorithms Implemented in Software	22
3. Analysis of Random Number Generators Implemented in Hardware	24
3.1 Previous Hardware Random Number Generators Implemented	24
3.2 Performance Measurements	26
3.3 Random Number Generator Implementations	27

3.3.1 Linear Feedback Shift Registers	27
3.3.2 Multiple Linear Feedback Shift Registers	29
3.3.3 Cellular Automata	29
3.3.4 Multiple Cellular Automata	30
3.4 Results	31
4. Implementation of Genetic Algorithm Modules and Architectures	34
4.1 Random Number Generators	34
4.1.1 Linear Feedback Shift Register	35
4.1.2 Multiple Linear Feedback Shift Register	36
4.2 Selection Modules	37
4.2.1 Roulette Wheel Selection	37
4.3 Crossover Modules	40
4.3.1 One-point Crossover	41
4.3.2 Two-point Crossover	41
4.3.3 Uniform Crossover	42
4.3.4 Arithmetic Crossover	43
4.4 Mutation Module	43
4.5 Fitness Modules	44
4.6 Example Architecture	45

5. Performance Analysis	55
5.1 GA Modules	55
5.2 Test Problems	56
5.3 Experimental Setup	57
5.4 Functional Verification	60
5.5 Performance Analysis	61
5.6 Comparison to Software-based GA	71
 6. Conclusions and Future Work	 74
Bibliography	77

List of Figures

Figure 2.1.	The Various Processes of a Genetic Algorithm	14
Figure 2.2.	A Simple Genetic Algorithm	15
Figure 2.3.	Flowchart of a Simple Genetic Algorithm	16
Figure 2.4.	Flowchart of a Steady-State Genetic Algorithm	19
Figure 3.1.	8-bit LFSR	27
Figure 3.2.	32-bit LFSR Used by Martin	28
Figure 3.3.	Serial Test of a Simple LFSR RNG	28
Figure 3.4.	Serial Test for a RNG Using 16 Parallel LFSRs	29
Figure 3.5.	Serial Test for One-Dimensional CA RNG	30
Figure 3.6.	Serial Test for 32 CA RNG	31
Figure 4.1.	8-bit Linear Feedback Shift Register	36
Figure 4.2.	Roulette Wheel Representation for an Example Population of 4 Chromosomes	38
Figure 4.3.	One-Point Crossover	41
Figure 4.4.	Two-Point Crossover	42
Figure 4.5.	Uniform Crossover	42
Figure 4.6.	Arithmetic Crossover	43
Figure 4.7.	Mutation Operation	44
Figure 4.8.	Architecture for the GA System Implemented in This Work	46

Figure 4.9	Genetic Algorithm System Architecture	48
Figure 4.10.	Genetic Algorithm Architecture Implemented for Traveling Sales- man Problems	49
Figure 4.11.	Genetic Algorithm Architecture	51
Figure 4.12.	General Purpose Genetic Algorithm	52
Figure 5.1.	Architecture of the GA System Implemented in This Work	58
Figure 5.2.	Maximum Value Obtained for The Initial Population Generated by the LFSR Based RNG (16 runs)	63
Figure 5.3.	Maximum Value Obtained for the Fitness Function $f(x) = 2x$ Implemented Using Multiple LPSR Based RNG	65
Figure 5.4.	Comparison of Results from the LFSR Based Genetic Algorithm Implementations and Multiple LFSR Based GA Implementations	66
Figure 5.5.	Performance Analysis for the 8-bit GA Systems Maximizing the Function $f(x) = 2x^3 - 45x^2 + 300x$	68
Figure 5.6.	Results for the different GA Implementations for $f(x) = 2x^3 - 45x^2 + 300x$ (Using Multiple LFSR Based RNG)	70
Figure 5.7.	Comparison between LFSR Based RNG Systems and Multiple LFSR Based Systems for the Function $f(x) = 2x^3 - 45x^2 + 300x$	71
Figure 5.8.	MATLAB Simulation Results for $f(x) = 2x$	72
Figure 5.9.	MATLAB Simulation Results for $f(x) = 2x^3 - 45x^2 + 300x$	73

List of Tables

Table 3.1.	Diehard Test Results for the Different RNGs	32
Table 3.2.	Summary of RNG Performance When Implemented on a FPGA	32
Table 4.1.	Example Population to Illustrate Roulette Wheel Selection	39
Table 4.2.	Individuals Selected for the GA Population Shown by the Roulette Wheel Selection Method	39
Table 5.1.	The Different GA Modules Implemented in Hardware and the Version Of the MATLAB Implementation from [19] Which Is Compared to Them	56
Table 5.2.	Performance Results for the Genetic Algorithm to Optimize the Function $f(x) = 2x$.	62
Table 5.3.	Results for the Function $f(x) = 2x$ Using Multiple LFSR Based RNG	64
Table 5.4.	Results for the GA Implementations Using LFSR Based RNG to Minimize the Fitness Function $f(x) = 2x^3 - 45x^2 + 300x$	67
Table 5.5.	Performance Analysis for the 8-bit GA Systems Maximizing The Function $f(x) = 2x^3 - 45x^2 + 300x$	69

Chapter 1

Introduction

Genetic Algorithms (GAs) are robust search and optimization algorithms that mimic the theory of evolution and natural selection. GAs were originally developed by John Holland in 1975 [1] and have since been applied to many applications, including the Traveling Salesman Problem (TSP) [5], and circuit partitioning problems [8].

Since GAs are heuristic methods, they are not guaranteed to find the optimum, but various applications [5, 7, 11, 10] have shown that they are able to find very good solutions to hard optimization problems. But GAs implemented in software, when applied to increasingly complex problems and large search spaces, can cause unacceptable delays. Hence, an alternative to this approach is to implement the GA in hardware. The combination of pipelining and parallelism possible in a hardware GA implementation increases the speed of the system when compared to its software counterpart. This allows the hardware implementation to be applied to various complex applications.

Since GA operations are relatively simple, GAs are good candidates for FPGAs and reconfigurable systems. This thesis deals with the implementation of a GA system in

hardware, where the various modules of the GA system are described in Hardware Description Languages like Verilog HDL and VHDL. These HDL modules can be incorporated into various architectures like the Island model and the Master-Slave model [24] in order to fully exploit the features of pipelining and parallelization. The HDL modules can be used by other researchers to implement GAs in hardware and can also be incorporated into efficient architectures for specific problems of interest. These hardware modules can be further synthesized and mapped to Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs).

1.1 Motivation

GAs are extensively used in many applications across a large and growing number of disciplines. Although a GA is able to find very good solutions for a variety of applications, the amount of time consumed for large computations and iterations is enormous. Hence, software implementation of GAs for increasingly complex applications can cause unacceptable delays. Also, GAs lend themselves easily to pipelining and parallelization. All these factors make GAs good candidates for hardware implementation. This thesis presents the design of a library of modules to support hardware implementation of GAs, providing a rich set of modules and example architectures which users can easily be customized for specific applications.

1.2 Introduction to Genetic Algorithms

Genetic Algorithms are a class of evolutionary algorithms that use biologically-derived techniques such as inheritance, natural selection, crossover (or recombination) and mutation [1]. They work on the principle of “survival of the fittest”, where the less fit members of a particular generation are replaced by new members formed by combining parts of highly fit members. Traditionally, solutions to GAs are represented in binary as strings of 0s and 1s, but different encoding schemes are also possible.

GAs differ from other conventional optimization processes in the following ways [1]:

- GAs work with a coding of the parameter set, not the parameters themselves;
- GAs search from a population of points, not a single point;
- GAs use objective function information, not derivatives or other auxiliary knowledge;
- GAs use probabilistic transition rules, not deterministic rules.

Genetic Algorithms work by evolving a population of members over a number of generations. First, the initial population of members is generated. After this, the fitness of each member is evaluated, where the fitness is a function of the application. Two members or individuals are then selected simultaneously. The selection is usually proportional to the fitness value, i.e., the members with higher fitness have a greater probability of being selected. These members are crossed to form new individuals and these new members are occasionally mutated to avoid convergence to local optima. The resulting members replace the less fit members of the old population. This process is

repeated until a stopping criterion is met. The stopping criterion could be either a predetermined number of generations or lack of improvement in the fitness values after a certain number of generations. Thus, the fitness value of the population is typically improved with every new generation.

1.3 Introduction to Verilog HDL and VHDL

Hardware Description Languages (HDLs) like Verilog HDL and VHDL (Very High Speed Integrated Circuits HDL) were developed as a means to allow designers to model the concurrency of processes found in hardware elements [2]. HDLs are now widely used for logic verification and synthesis. A typical ASIC design flow begins with the description of the design at the Register Transfer Level (RTL) using HDLs like Verilog HDL and VHDL. Logic synthesis tools are then used to convert this RTL description to the gate-level netlist, which describes the circuit in terms of the gates and connections between them. The synthesis tools ensure that the gate-level netlist meets the area, timing and power constraints. This netlist is the input to the Automatic Place and Route tool that creates a layout. Upon verification of the layout, the chip can be fabricated.

Verilog HDL and VHDL have many advantages over the traditional schematic-based design. Some of these are [2]:

- designs can be chosen at a very abstract level by using HDLs. The RTL descriptions can be written without choosing a specific technology. The synthesis tools can automatically convert the design to any fabrication technology.
- functional verification of the design described at the RTL level can be done early in the design cycle. The design can be optimized and modified to meet the desired functionality. This reduces the design cycle time significantly.
- designing with HDLs is analogous to computer programming. A textual description is easier to use to develop and debug circuits and it provides a concise representation of the design. Gate-level schematics are almost incomprehensible for very complex designs.

1.4 Organization of the Thesis

The thesis is organized as follows. Chapter 2 presents the fundamentals of GAs, explaining the basic terminology and illustrating a simple GA example. This chapter also describes previous work on GAs implemented in hardware and software. Chapter 3 deals with an analysis of the Random Number Generators (RNGs) implemented in hardware. Different RNGs are compared and contrasted. Chapter 4 describes the various GA modules that are implemented in this work and example architectures that could be used by other researchers for application-specific GA systems. Chapter 5 analyses the simulation results and the performance analysis of the modules implemented. Finally, chapter 6 presents a summary of this work and discusses possible avenues for future work.

Chapter 2

Overview of Genetic Algorithms

Genetic Algorithms were developed by John Holland in 1975 in an attempt to explain the adaptive processes of natural selection and to thus design an artificial system's software to retain the important mechanisms of natural systems [1]. A typical GA is an intelligent search strategy supported by operations inspired by biological evolution. The theory of evolution was introduced by Charles Darwin to explain his observations of plants and animals in the natural ecosystem [4]. He observed that as every new generation was associated with some changes, the less-fit individuals tended to lose the battle for survival in the competition for food. Thus, this principle of "survival of the fittest" leads to improvement in the species with every new generation and this was the basic principle implemented in all GA systems.

Genetic Algorithms are heuristic search algorithms based on natural genetics. They represent an intelligent exploitation of random search to solve optimization problems. Although randomized, GAs are not totally random, since they exploit historical information to direct the search into regions of better performance within the search space [18].

2.1 Terminology of Genetic Algorithms

Because Genetic Algorithms are rooted in both natural genetics and computer science, the terminology used in the GA literature is a mix of the natural and artificial [1]. All GAs work on a *population* or a collection of solutions to the given problem [4]. Each individual in the population is called a string or *chromosome*, in analogy to chromosomes in natural systems. These individuals are coded as binary strings and the individual characters in the strings are referred to as *genes*. In each iteration of the GA, a new generation is evolved from the existing population in an attempt to obtain better solutions.

The *fitness* function i.e., the evaluation function is used to determine the fitness of each chromosome. The fitness function is problem specific and user defined. In the selection process, two chromosomes are chosen simultaneously from the population for reproduction. These selected strings are called *parents*. Crossover is the main operator used for reproduction. It combines properties of both parents to create two new chromosomes called *offspring*.

Mutation is an incremental change made to each member of the population, with a very small probability. Mutation enables new features to be introduced into a population. The various processes of the GA system are shown below [figure 2.1].

Figure 2.1. The Various Processes of a GA System [18]

GAs basically use two types of population replacement schemes:

- a total replacement scheme or the simple GA
- a partial replacement scheme or the steady state GA.

These are described in detail in the following sections.

2.2 A Simple Genetic Algorithm

Two of the most compelling reasons for using a GA are simplicity of operation and power of effect [1]. The simple GA is also known as the total replacement algorithm and this is illustrated in figure 2.2.

Figure 2.2. A Simple Genetic Algorithm [4]

A simple GA is basically composed of three operators – selection, crossover and mutation. These three operators primarily involve random number generation, copying and swapping partial strings, thus making these GAs simple to implement. Figure 2.3 below illustrates the simple GA process in flowchart form.

Figure 2.3. Flowchart of a Simple Genetic Algorithm [4]

The GA process starts with the generation of the initial population, which is usually generated randomly, but may also be supplied by the user. Each chromosome of the population is evaluated according to some fitness function that is application specific. This is followed by the selection process where two individuals are selected at a time. These two parents then undergo crossover to form two new offspring, which are then mutated with a low mutation probability. A highly fit population is evolved through several generations of selection, crossover and mutation until some stopping criteria are met [4].

The GA can be terminated according to several stopping criteria:

- a fixed number of generations have occurred;
- all individuals in the population converge to the same string;
- no improvements in the fitness value are found after a certain number of generations.

Once the stopping criterion is met and the new generation is evolved, the old generation can be discarded.

2.3 The Steady-State Algorithm

The steady-state GA is also known as the partial replacement algorithm because only a fraction of the individuals are replaced in each generation [4]. The steady-state GA process begins with the generation of the initial population, after which each individual is evaluated according to some fitness function. Based on the fitness, two different

individuals are selected at a time. These parents are crossed over to form two new offspring. The offspring are then mutated with a low probability of mutation. The mutated offspring are evaluated and compared to the worst individuals in the population. If they are better than the worst individuals, they replace them (the worst individuals), else they are discarded. This completes the generation. In the steady-state GA, the generation gap is minimal since only two offspring are produced in each generation. Figure 2.4 illustrates the partial replacement algorithm.

Figure 2.4. Flowchart of a Steady-state Genetic Algorithm [4]

The steady state GA tends to stagnate as the majority of the offspring may be inferior. Thus the GA rejects them, making more trials on the existing population without any gain.

2.4 Previous Work in Hardware Genetic Algorithms

The past several years have seen a number of Genetic Algorithms implemented in hardware. This section gives a brief overview of the various GA implementations by different researchers in the last few years. Schematics for these implementations can be found in Chapter 4.

Koonar et al developed a reconfigurable GA for VLSI CAD design. This work described a GA architecture for circuit partitioning in VLSI physical design automation. The design used 6 modules along with three external memories. This design was synthesized on Virtex part xcv 50e using Xilinx ISE 4.1. This hardware implementation achieved a speedup of 100x over its software counterpart [8]. This design was implemented in VHDL.

Tommiska et al realized a general purpose GA in Altera HDL (AHDL). The design was simulated with programmable logic devices of Altera's Flex 10k Field Programmable Gate Array (FPGA) family. The GA was run in a pipelined fashion that enabled it to be

212 times faster than the software solution [11]. This GA system was implemented in Altera HDL (AHDL).

Graham et al implemented a Splash 2 Parallel GA (SPGA) for optimizing symmetric traveling salesman problems in VHDL. Each processor in the SPGA consisted of four FPGAs and associated memories. This system was found to be 6 – 10 times faster than the equivalent software version [5, 10]. This GA system was based on the Simple GA scheme.

Scott et al described a general purpose Hardware Genetic Algorithm (HGA) to be used in many applications where conventional GAs were too slow. This design was realized using VHDL and implemented on a BORG prototyping board which consisted of five Xilinx FPGAs. Various HGA implementations were described here which exploited parallelism and coarse-grained pipelining [6]. This GA implementation was described in VHDL and was based on the Simple GA or total replacement scheme.

Shackleford et al developed a survival-based, steady state GA that was aimed at achieving high performance. The prototype GA machine was designed using the Tsutsuji logic synthesis system and implemented on an Aptix AXB-MP3 Field Programmable Circuit Board (FPCB) populated with six FPGAs [7]. This implementation was described in VHDL.

In contrast to the simple GA, Aportewan et al implemented a compact GA in Verilog HDL as they claimed this was more suitable for hardware implementation. This system was found to be 1000x faster than the software version [9].

2.5 Genetic Algorithms Implemented in Software

There are many implementations of GAs in software. Here we concentrate on one widely used implementation, the Genetic Algorithm Optimization Toolbox (GAOT), implemented in MATLAB [19]. This toolbox is widely used and thus is a good basis for defining our hardware module library. The GAOT is a Genetic Algorithm system that is employed for function optimization using binary and real representations. MATLAB was used for the implementation as it provided many built in auxiliary functions useful for function optimization and also because it is useful for numerical computations. The toolbox constitutes a library of GA modules such as Roulette Wheel, Tournament selection and Normalized Geometric selection for the selection process; One-point, Arithmetic and Heuristic methods for crossover; and various mutation methods. Most of these methods are discussed in detail in Chapter 4. The Normalized Geometric selection and the Heuristic crossover method are used for real representation and are not implemented in this thesis work. Floating point representations have not been dealt in this thesis work and are discussed in the future work section.

The toolbox has been tested on a series of non-linear, non-convex and multi-modal functions. The results show that the algorithm finds better solutions with less function evaluations than simulated annealing [19].

Chapter 3

Analysis of Random Number Generators Implemented in Hardware

The Random Number Generator (RNG) is a basic component of Genetic Algorithms (GAs) as it generates the initial population and also provides inputs to the selection modules, the crossover modules and the mutation module [26]. Since these random numbers generated are computed by deterministic algorithms, they are more accurately called Pseudo Random Number Generators (PRNGs). Here, an analysis of the performance of the PRNGs implemented in hardware is presented and the effect of PRNGs on a simple GA is discussed, according to the results given in [20, 23].

3.1 Previous Hardware Random Number Generators Implemented

Previous work [23] indicated that the quality of the pseudo random number generator has little effect on the performance of a GA system. Here, an analysis of the effects of using different random number generators in hardware implementation of GAs is presented as given in [23, 20]. The HGAs mentioned in the previous section (section 2.4) have

typically used RNGs based on Linear Feedback Shift Registers (LFSR), Linear Shift Register (LSHR) or Cellular Automata (CA).

In the general purpose Hardware Genetic Algorithm (HGA) developed by Scott et al, pseudo random numbers were generated through the use of cellular automata [6]. The CA used in the RNG was used to generate 4-bit numbers and consisted of 16 alternating cells that change their states according to rules 90 and 150 given below:

$$\text{Rule 90: } S_i^+ = S_{i-1} \text{ xor } S_{i+1}$$

$$\text{Rule 150: } S_i^+ = S_{i-1} \text{ xor } S_i \text{ xor } S_{i+1}$$

Here, S_i is the current state of the cell i in the linear array, S_i^+ is the next state for S_i . Thus, rule 90 updates itself based on only inputs from its neighbors while rule 150 considers its own state along with its neighbor states while updating. The structure of the RNG described here resembles that of a LFSR but has the advantage of having only local interconnects between flip flops, making the CA version operate at higher clock frequency.

Graham et al [8] also implemented the PRNG based on Scott's [6] design. Shackleford et al [7] implemented a CA based on the work of Wolfram [25]. Aportewan et al [10] used a one-dimensional two-state CA to produce random numbers. Matti et al [11] used a LSHR based RNG in the HGA implementation. Three shift registers with appropriate lengths were used so as to attain long periodicity of the pseudo random numbers. Also, the seed to this LSHR was generated with the help of a noise diode, an amplifier and a high-speed

(10 mega samples/sec) 12-bit Analog to Digital converter. This approach was used to attain true randomness for the seed. An evaluation of the different properties of these random number generators is presented here.

3.2 Performance Measurements

The performances of the various RNG implementations were tested by Martin [20] using three methods:

- he used the Diehard test suite maintained by Marsaglia [22] to gauge the general performance of the RNG. This suite consisted of 15 tests that were modeled on applications of random numbers. All the RNGs were implemented by Martin in ISO-C and submitted to all the 15 tests. Each Diehard test produces one or more p-values. A p-value ≥ 0.998 is considered as bad, $0.95 \leq p < 0.998$ is considered as suspect and all other p-values are classified as good. Each bad p-value scores a 4, suspect scores a 2 and each good p-value scores a zero. Thus, the scores for each test were summed to obtain the total for score for each RNG. This score scheme indicated a poor RNG if the score was high and a good RNG for low scores.

- he used each RNG that was implemented in the artificial ant problem with the Santa Fe trail. The problem was run 500 times to record the number of correct programs, which gave a measure of the performance of the RNG. The population size was fixed at 1024, the maximum program length was 31 and all experiments were run for 32 generations.

- he also implemented each RNG as a stand alone application for an FPGA so as to record the number of slices used and the maximum clock frequency attained. This gave a measure of the hardware resources needed to implement the RNG and also the logic depth required.

3.3 Random Number Generator Implementations

3.3.1 Linear Feedback Shift Register

The most common way to implement a PRNG is a LFSR. The basic 8-bit LFSR is shown in figure 3.1. It is basically a series connection of flip flops with XOR feedback, where the XOR gate is used to scramble the input bit [13]. The feedback taps for LFSRs of different lengths are given in [13] and these are used in order to get the maximum length sequence of random numbers $2^n - 1$.

Figure 3.1. 8-bit LFSR [16]

A 32-bit LFSR that was tested for performance by Martin is shown below [fig 3.2.].

Figure 3.2. 32-bit LFSR used by Martin [20]

This type of RNG fails the sequential test described by Knuth [21]. This is because, at any time t , there is a 50% probability that the value at time $t+1$ can be predicted. For an LFSR of length n at time t if the value is v , then at time $t+1$ the value will be $v/2$ or $(v/2 + 2^{n-1})$. Figure 3.3. shows the plot of values v_t and v_{t+1} and it can be seen that the random numbers follow a pattern and are predictable.

Figure 3.3. Serial Test of a Simple LFSR RNG [20]

3.3.2 Multiple Linear Feedback Shift Registers

To improve the performance of the LFSR discussed above, n LFSRs of length m are implemented such that a single bit from each LFSR is used at each step time. Martin [20] studied the performance of 32 16-bit LFSRs that run in parallel, where each LFSR was initialized to a different value. The serial test for this kind of implementation is found to be better than the previous implementation as shown in figure 3.4 below. The figure shows that the numbers generated are random and they do not follow a regular pattern.

Figure 3.4. Serial Test for an RNG Using 16 Parallel LFSRs [20]

3.3.3 Cellular Automata

Cellular Automata are groups of cells where each cell's life (state) depends on its neighboring cells. The state of the cell in the next cycle is determined by following simple rules that check the state of the cell itself and the state of its neighbors [17].

In the one-dimensional CA RNG described in Martin's paper [20], each cell has two neighbors – left and right (sometimes called west and east respectively). For each cycle, the state or value of any cell is given by a rule. For the RNG tested here, rule 30 was used, which states that for any cell c at time t , $c_{t+1} = (\text{west}_t + c_t) \text{ xor east}_t$. The results for the serial test are shown in figure 3.5. It was observed that there was a distinct pattern to the numbers as in the case of LFSR [20].

Figure 3.5. Serial Test for One-Dimensional CA RNG [20]

3.3.4 Multiple Cellular Automata

For better performance results, several CAs are combined for the implementation. For this test, 32 CAs were implemented and one bit from each CA was taken to generate a 32-bit random number. The serial test result is shown below in figure 3.6, and it can be seen that it is more random than the one-dimensional CA described above [20].

Figure 3.6. Serial Test for 32 CA RNG [20]

3.4 Results

The results of running the Diehard tests for the RNGs described above are shown in table 3.1. From the results obtained by Martin, it is observed that the multiple-LFSR implementation performed the best. The Linear Shift Register implementation is referred to as LFSR, the multiple LFSR is referred to as 32LFSR, the one-dimensional CA is referred to as 1DCA and the multiple CA is referred to as 32CA in table 3.1.

Test	Max score	LFSR	32LFSR	IDCA	32CA
Birthday	36	36	2	0	8
Overlapping permutation	8	8	4	8	8
Binary Rank 32x32	8	8	8	2	6
Binary Rank 6x	104	104	8	140	70
Bitstream	80	80	0	80	80
Overlapping pairs tests	328	328	94	328	320
Count the ones (stream)	8	8	8	8	8
Count the ones (specific)	100	100	30	100	100
Parking Lot	44	4	0	4	2
Minimum Distance	4	4	4	4	4
3D spheres	84	4	2	4	2
Squeeze	4	4	0	4	4
Overlapping Sums	44	44	0	6	0
Runs	16	16	2	16	8
Craps	8	8	0	8	12
Total	876	756	162	676	640

Table 3.1. Diehard Test Results for the Different RNGs [20]

RNG	Rank	Correct	% Correct	Slice	Clock rate F_{max} (MHz)
32CA	1	190	38.0%	284	105
32LFSR	3	185	37.0%	130	134
ID CA	5	182	36.4%	22	125
LFSR	6	159	31.8%	18	188

Table 3.2. Summary of RNG Performance When Implemented on a FPGA [20]

Martin [20] also investigated RNG performance for the artificial ant problem. The number of correct programs that were produced by each RNG for the artificial ant problem is shown in table 3.2. This table also shows the slice count and maximum clock frequency of the RNGs implemented on the FPGA.

Despite the weakness of the simple LFSR and the simple CA RNG, the total effect on the performance of the GA that used multiple LFSRs and CAs appeared to be very small. Also, the slice count of the multiple-CA implementation was considerably higher than the other RNG implementation. Thus, the implementation used more FPGA resources without any considerable improvement in performance.

Hence, it was concluded by Martin and Meysenburg et al in [20, 23] that complicated RNG implementation which used more FPGA resources had very little effect on the overall performance of the GA system.

Chapter 4

Implementation of Genetic Algorithm

Modules and Architectures

This chapter describes the various modules of the Genetic Algorithm system. These GA modules are implemented in Verilog HDL and VHDL and consist of the different types of Random Number Generators, Selection modules, Crossover modules, Mutation module and Fitness modules. This chapter also describes several architectures that the modules could be incorporated into, which exploit the features of pipelining and parallelism.

4.1. Random Number Generators

Based on the analysis of the Random Number Generators (RNGs) and the results obtained by Martin [20] discussed in chapter 3, the Linear Feedback Shift Register and the Multiple LFSR are implemented in this thesis. Though the RNG implementations based on the theory of Cellular Automata were also described in the previous chapter, these implementations do not obtain great performance improvement over the LFSR implementations. The CA implementation is a sophisticated implementation which

requires more FPGA resources and occupies more area; hence this was not considered for implementation in this thesis work.

4.1.1. Linear Feedback Shift Register

A Linear Feedback Shift Register is a sequential shift register with combinational logic that causes it to generate pseudo random numbers. The implemented LFSR has speed and area advantages over other implementations. Feedback for the LFSR is provided by a selection of points called taps in the register chain. These points or taps are Exclusive-ORed (XORed) to obtain the feedback. It is this feedback that causes the register to loop through repetitive sequences of pseudo-random values. The implemented LFSR is coded for a maximum sequence length of 2^{n-1} , where ' n ' is the number of bits in the LFSR [12].

The LFSR has one major drawback; if all the registers happen to be '0', the shift register gets "stuck". This is because the XOR of all '0's is a '0' and hence the XOR feedback does not produce a '1' to get the sequence started again. To prevent this condition, the routine must be first loaded with a non-zero seed value. This value can be any number as long as it is a non-zero number. The numbers generated by the LFSR are based on the seed value and the same sequence of numbers is repeated until the LSFR is reloaded with a different seed [13].

Figure 4.1 shows an 8-bit Linear Feedback Shift Register. In this thesis, an 8-bit and a 16-bit Linear Feedback Shift Register is implemented, where the seed value is given by the user.

Figure 4.1. 8-bit Linear Feedback Shift Register [12]

4.1.2 Multiple Linear Feedback Shift Registers

In this implementation, multiple LFSRs are implemented such that one bit from each LFSR is used to form a random number. Each of these LFSRs is initialized with a different value. For an 8-bit random number, eight 8-bit LFSRs are implemented and 1-bit from each LFSR is taken to form the required random number. This implementation has better performance results than the previous one, as shown by Martin [20]. Here, an 8-bit and a 16-bit multiple LFSR has been implemented.

4.2. Selection Modules

There are various selection schemes for Genetic Algorithms, each with different characteristics. An ideal selection scheme is one that is easy to code, and efficient for both non-parallel and parallel architectures. Furthermore, a selection scheme should be able to adjust its selection pressure to adapt to different domains, where the selection pressure is the degree to which better individuals are favored [14].

The main task of the selection module is to select individuals from the population so that these individuals can be sent to the crossover and mutation modules in order to attain new offspring. There are numerous selection methods such as roulette wheel selection, rank selection, tournament selection etc., and they are usually biased towards more highly fit individuals. Selection is one of the key operators on GAs that ensures survival of the fittest [4]. The different types of selection modules implemented in this thesis work are described below:

4.2.1. Roulette Wheel Selection

This is one of the more popular fitness-proportional selection methods. Here, the parents or individuals are selected based on their fitness values, and thus more fit chromosomes have a higher probability of being selected. The analogy to a roulette wheel can be envisaged by imagining a roulette wheel in which each candidate solution represents a section on the wheel; the size of the sections are proportionate to the appropriate fitness

measure [14]. The size of the section also represents the probability of selection of the solution, thus the larger the section the greater the chance of the chromosome being selected. Selecting N chromosomes from the population is equivalent to playing N games on the roulette wheel, as each candidate is drawn independently.

The following figure represents an example.

Figure 4.2. Roulette Wheel Representation for an Example Population of 4 Chromosomes [27]

For each selection, the wheel is spun and a random position on the wheel is chosen and the chromosome that represents that position is selected. This clearly shows that the chromosomes with bigger fitness values will be selected more often.

This process can be described by the following algorithm:

- For each chromosome, the fitness and the cumulative fitness are evaluated. This step is performed only once for each generation.
- A random number 'r' is generated.

- The chromosome whose cumulative fitness is the first value greater than the generated random number 'r' is selected as the “parent” chromosome.

An example of the Roulette Wheel Selection for a GA process is illustrated below. This process has an initial population of six members.

Individual	Chromosome	Fitness	Cumulative Fitness
X1	101100	20	20
X2	111000	7	27
X3	001110	6	33
X4	101010	10	43
X5	100011	12	55
X6	011011	9	64

Table 4.1. Example Population to Illustrate Roulette Wheel Selection

Random Number Generated	Individual Selected
24.3	X2
42.8	X4
54.88	X6
28	X2
19.78	X1

Table 4.2. Individuals Selected for The GA Population by the Roulette Wheel

Selection Method

4.3. Crossover Modules

Crossover is the fundamental mechanism of genetic rearrangement in Genetic Algorithms. The purpose of crossing individuals in GAs is to test new parts of the target regions instead of testing the same string repeatedly in successive generations [15]. Once the Selection Module selects two chromosomes, the Crossover Module is used to generate two new offspring. This is mainly done by the exchange of strings between the two parent chromosomes. If the best strings of the parents are combined then the resultant offspring may be better than the two parents. The amount of crossover is controlled by the crossover probability, P_c , which is defined as the ratio of the number of offspring produced in each generation to the population size. A higher crossover probability allows proper exploration of the search space and reduces the chances of settling for a false optimum while a lower crossover probability enables exploitation of existing individuals in the population that have relatively high fitness [4]. Thus, the performance of the GA depends greatly on the type of crossover operator used.

There are many crossover schemes such as the one-point crossover, two-point crossover, uniform crossover, arithmetic crossover etc., and these are dealt with in detail in the next few sections.

4.3.1. One-point Crossover

In this crossover scheme, a single crossover point is chosen at random and the first offspring is equal to the first parent up to the crossover point and identical to the second parent after the crossover point. The second offspring is identical to the second parent till the crossover point and to the first parent after the crossover point.

This scheme is illustrated with an example in figure 4.3. '*Parent1*' and '*Parent2*' represent the two parent chromosomes and a crossover point between bits 3 and 4 is selected. The two resulting offspring obtained after the process of crossover are also shown.

	7	6	5	4	3	2	1	0	bit position
Parent1:	1	0	1	0	1	0	1	1	
Parent2:	1	1	0	1	1	1	1	1	
Offspring1:	1	0	1	0	1	1	1	1	
Offspring2:	1	1	0	1	1	0	1	1	

Figure 4.3. One-point Crossover

4.3.2. Two-point Crossover

The Two-point Crossover has two crossover points and produces two offspring. The first offspring consists of strings from the first parent until the first crossover point, and then the strings from the second parent are copied till the second crossover point and the remaining is copied from the first parent again. This process is illustrated with an

example, where parent1 and parent2 are the two parent chromosomes and offspring1 and offspring2 are the two offspring. The two crossover points are between the 2nd and 3rd bits and between the 5th and 6th bits (figure 4.4).

```

          7 6 | 5 4 3 | 2 1 0 bit position
Parent1:  1 0 | 1 0 1 | 0 1 1
Parent2:  1 1 | 0 1 1 | 1 1 1
Offspring1: 1 0 | 0 1 1 | 0 1 1
Offspring2: 1 1 | 1 0 1 | 1 1 1

```

Figure 4.4. Two-point Crossover

4.3.3. Uniform Crossover

Here, each bit of the offspring is selected randomly from the corresponding bits of either the first parent or the second parent. This crossover scheme produces only one offspring, unlike the two crossover operators discussed above. An example of the uniform crossover operator is shown in figure 4.5. Parent1 and parent2 represent the two parent chromosomes and offspring1 is the result obtained. Because only one offspring is produced, a smaller population set is obtained after the GA process.

```

          7 6 5 4 3 2 1 0 bit position
Parent1:  1 0 1 0 1 0 1 1
Parent2:  0 1 0 1 1 1 1 0
Offspring1: 0 0 0 0 1 0 1 1

```

Figure 4.5. Uniform Crossover

4.3.4. Arithmetic Crossover

The arithmetic crossover operation performs some arithmetic or logical operations on the parent chromosomes to produce the offspring. Example arithmetic and logical operations are addition, subtraction, AND, OR, NAND, NOR, XOR etc. This crossover scheme is illustrated in figure 4.6., where the AND function is performed to obtain the offspring. This scheme also produces only one offspring, as does the uniform crossover operator.

	7	6	5	4	3	2	1	0	bit position
Parent1:	1	0	1	0	1	0	1	1	
Parent2:	0	1	0	1	1	1	1	0	
Offspring1:	0	0	0	0	1	0	1	0	

Figure 4.6. Arithmetic Crossover for AND Function

4.4. Mutation Module

Mutation is the process which consists of making small alterations to the bits of the chromosomes. In the binary-encoded GA, mutation is merely the process of flipping random bits of the chromosomes that have undergone crossover. The mutation process is performed with a small probability of mutation, P_m , which is defined as the probability of mutating each chromosome. It controls the rate at which new bit values are introduced into the population. If P_m is too high, the offspring will be too random and hence lose

their resemblance to the parents. Hence, low mutation rates are preferred so that new values are introduced that may prove to be useful.

The mutation process is illustrated in figure 4.7, where bit 4 undergoes mutation.

	7	6	5	4	3	2	1	0	bit position
Before Mutation:	1	0	1	1	0	1	1	1	
After Mutation:	1	0	1	0	0	1	1	1	

Figure 4.7. Mutation Operation

Although the crossover operator is the most efficient search mechanism, it does not independently guarantee exploration of the entire search space with a finite population size. This gap is filled in by the mutation operator [4].

4.5. Fitness Modules

The fitness module is application specific and is designed according to the function to be optimized or evaluated. In this thesis, two separate functions have been presented for optimization and these are based on functions implemented by Scott [6] in his thesis work. The two mathematical functions are given below:

(i) $f(x) = 2x$

(ii) $f(x) = 2x^3 - 45x^2 + 300x$

These functions are optimized by using different combinations of modules from the library developed in this work. The performances of the GA system for different implementations are compared in Chapter 5.

Different fitness functions can be implemented by researchers for specific applications and these fitness modules can be incorporated with other modules available in the library developed here to form an efficient GA system.

4.6. Example Architecture

This section describes the architecture that the GA modules were incorporated into. Several other architectures that exploit the features of pipelining and parallelization have also been described. These architectures have been implemented by other researchers for specific applications.

The various GA modules were connected together according to the architecture illustrated in figure 4.8. The initial population is generated by the Random Number Generator (RNG). This population is then fed into the Fitness Module (FM), which evaluates the fitness for the entire population set and then passes on the population members along with their fitness values to the Selection Module (SM). The SM selects

two parents based on the Roulette Wheel selection method and sends them to the Crossover Module (CM) which could be one of the four methods implemented in this thesis (one-point, two-point, uniform and arithmetic crossover). The resultant offspring from the CM are then mutated before being sent to the fitness module. The second FM evaluates the fitness of the final members and this population replaces the old population. This architecture is based on the Simple GA as it replaces the old population entirely.

Figure 4.8. Architecture for the GA system Implemented in This Work

Once the first set of parents are selected and passed over to the CM, the SM works on selecting the next set of parents while the crossover, mutation and fitness evaluation is being done. This pipelined system helps in increasing the overall speed of the system. The system also has two fitness modules, one to evaluate the fitness of the original population and another one to evaluate the fitness of the final population.

The following architectures have been designed by other researchers and could be used to incorporate the different GA modules from the HDL library developed in this thesis.

Koonar et al implemented a GA system for circuit partitioning problems in VLSI design. The architecture is shown in figure 4.9. It employs pipelining and parallelization to achieve speedup over the software counterpart. This system has six modules (control registers, main controller, selection module, fitness module, and crossover and mutation module) and three external memories (fitness memory, chromosome memory and netlist memory). The core process can be outlined as follows [8]:

- The selection module performs tournament selection on the initial population and selects four random fitness values from the fitness memory. It then compares two pairs of fitness values and selects the best from each pair. The addresses of the two parent chromosomes stored in the chromosome memory (corresponding to the fitness values selected) are latched and held stable on the output signal till the selection module is enabled again. The selection module contains a random number generator, a comparator, registers to latch the addresses and a control state machine.

Figure 4.9. GA System Architecture Described in [8]

- When the crossover and mutation module is enabled, it reads the parent chromosomes (supplied by the selection module) and performs crossover and mutation on them. The crossover and mutation rate is compared to an internally generated random number. If this number is less than the crossover and mutation rate, then it performs these operations otherwise the parent chromosomes are copied onto the next generation without any change. The resulting chromosomes obtained here are stored in the chromosome memory.

- The fitness module then evaluates the fitness of each of these chromosomes after the new population has been completely generated.
- The main controller is responsible for generating signals for all the blocks of the system. Upon receiving an active high, the controller reads the input netlist and loads this into the netlist memory. A LFSR is then employed to generate random numbers i.e., the initial population to the chromosome memory. After this the controller state machine enters a loop where it performs selection, crossover, mutation and fitness evaluation until a desired number of iterations are done. The main controller then enables the fitness module one last time and outputs the final new generation along with the fitness values.

A Splash2 Parallel Genetic Algorithm (SPGA) is implemented by Graham et al [5] to optimize solutions for Traveling Salesman Problems (TSPs). The problem involves finding the shortest path through n cities, visiting each city only once and returning to the starting point. The schematic for this system is given in figure 4.10.

Figure 4.10. GA Architecture Implemented for Traveling Salesman Problems [5]

This implementation has four FPGAs arranged in a pipelined fashion with associated memories. Each solution in the pipeline consists of an ordered list, referred to as '*tour*', which describes the sequence in which each city was visited. To optimize the TSP, the objective is to find the shortest tour, and so the fitness of a tour is associated with its length. Each FPGA of the SPGA performs functions as follows:

- FPGA1 performs selection based on the roulette wheel selection method and chooses tour pairs from memory. It passes the selected pair to FPGA2.
- FPGA2 performs crossover on the tour pairs. Depending on an internally generated random number, FPGA2 either performs crossover or copies the parent chromosomes without any change onto the next generation.
- FPGA3 evaluates the fitness value for the pair received from FPGA2. Then, it performs mutation on randomly selected tours. Finally, it sends the tours and fitness values to FPGA4.
- FPGA4 writes the new population into memory and records the best and worst fitness for that particular generation and the best fitness value to date.

These processes are repeated till the size of the new generation is equal to the original population size.

Matti et al [11] described an efficient GA architecture for a general purpose GA system. The pipeline consists of four stages which are separated by register banks, an essential component for synchronization of the pipeline and preservation of the address of the

chromosomes. This ensures that the same RAM addresses from which the chromosomes are read are also written into after the chromosomes are processed and evaluated.

The block diagram of the system designed is shown in figure 4.11. The RNG is implemented using linear shift registers (LSHR).

Figure 4.11. GA Architecture Implemented in [11]

The basic process of the GA system shown in figure 4.11 can be outlined as follows:

- At the 1st stage, two chromosomes are selected at random from the 32x32 bit RAM, which is implemented as a synchronous memory with separate read and write ports. The control register comes into play here as there are no bidirectional data busses.

- The selected chromosomes then undergo crossover and mutation. These offspring along with the original chromosomes are sent to the next stage.
- At the 3rd stage, the four chromosomes are evaluated. Each chromosome is compared to every other chromosome (i.e., a total of 6 comparisons). The two best chromosomes are selected and sent to the next stage. The fitness function here is a simple comparison between unsigned 32-bit binary numbers. This fitness function can be altered without affecting the overall speed of the system.
- The 2 chromosomes are written back into the same address from which the original chromosomes were read.

This pipeline is active during four consecutive cycles and inactive during the next four cycles, thus producing an efficiency of 50%.

A general purpose GA system that utilized a coarse-grained pipeline has been designed by Scott et al. The schematic of the GA architecture is shown in figure 4.12.

Figure 4.12. General Purpose GA Architecture [5]

The front end of the GA system consists of a simple interface program running on a PC or a workstation. This interface fetches the GA parameters (the initial population size and its members, the number of generations, the initial seed for the pseudo random number generator and the mutation and crossover probabilities) from the user and writes it into memory which is shared with the back end, comprising the Hardware Genetic Algorithm (HGA). When these values are received, the front end sends a 'go' signal to the back end, thus starting the GA run based on the GA parameters. When the run is completed, the back end sends a 'done' signal to the front end. The function of this module can be varied according to the physical memory attached to it and the desired interface between the HGA and the user. The basic function of the HGA design is as follows:

- Once all the parameters are loaded into the shared memory, the memory interface module (MIM) receives a 'go' signal from the front end. The MIM acts as the main controller of the HGA and is the only interface with the outside world.
- The MIM notifies the crossover and mutation module (CMM), the fitness module (FM), the pseudo random number generator (RNG) and the population sequencer (PS) that the HGA is ready to be executed. These modules fetch the required parameters from appropriate places in memory.
- The PS starts the pipeline by requesting population members from the memory and passing them onto the SM.
- The selection module selects a pair of chromosomes, based on the roulette wheel selection, from the set of members provided by the PS. Then, it sends the pair to the CMM after which the SM is reset and the selection process is repeated.

- The CMM performs crossover and mutation on the selected pair depending on a randomly generated value provided by the RNG. Then these offspring are sent over to the FM.
- The FM evaluates the two chromosomes and writes the new members along with their fitness values to the appropriate locations in memory through the MIM.

These steps are performed until the FM determines that the current HGA run is finished.

The stopping criterion here, is the number of generations.

Chapter 5

Performance Analysis

5.1 GA Modules

In this thesis work, various GA modules have been implemented in both Verilog HDL and VHDL. The different modules that constitute the HDL library are: Linear Feedback Shift Register (LFSR) and Multiple LFSR based Random Number Generators, Roulette Wheel Selection Module, Arithmetic, Uniform, One-point and Two-point Crossover Modules and a Mutation Module. The Fitness Module is designed according to the GA application. Here, the GA system is used to maximize the two functions $f(x) = 2x$ and $f(x) = 2x^3 - 45x^2 + 300x$. A complete list of the modules implemented in hardware is given in table 5.1. This table also lists the modules available in the GAOT toolbox implemented in MATLAB [19]. Only the modules for binary representation from the GAOT toolbox have been included in the table [table 5.1.]. The MATLAB implementation has other modules for real number representation, but these have not been included in this thesis work. The VHDL and Verilog HDL modules have only been implemented in binary, floating point implementations are mentioned in future work (chapter 6). Various combinations of the modules are incorporated into the GA architecture and the results from these implementations are compared to each other. The performance of the hardware implementations are also compared to the software GA implementation.

Modules	8-bit GA system	16-bit GA system	MATLAB implementation
Random Number Generators	(i) LFSR based RNG (ii) Multiple LFSR based RNG	(i) LFSR based RNG (ii) Multiple LFSR based RNG	MATLAB function <i>rand</i>
Fitness Function	Maximizing a given function	Maximizing a given function	Maximizing a given function
Selection Modules	Roulette Wheel selection	Roulette Wheel selection	Roulette Wheel selection
Crossover Modules	(i) Arithmetic crossover (ii) Uniform crossover (iii) Two-point crossover (iv) One-point crossover	(i) Arithmetic crossover (ii) Uniform crossover (iii) Two-point crossover (iv) One-point crossover	(i) One-point crossover
Mutation Modules	One-point mutation	One-point mutation	One-point mutation

Table 5.1. The Different GA Modules Implemented in Hardware and the Version Of the MATLAB Implementation from [19] Which Is Compared to Them

5.2 Test Problems

The different GA modules implemented in Verilog HDL and VHDL were employed to form a GA system that was used to maximize the given functions. The GA implementation in the MATLAB environment was also used to maximize the same

functions. The GA system shown in figure 5.1 was tested to maximize two mathematical functions and these functions are similar to the ones used in Scott's [5] thesis. The functions are given below and these were implemented as the fitness modules.

(i) $f(x) = 2x$

(ii) $f(x) = 2x^3 - 45x^2 + 300x$

The performance analyses of the GA system for different combination of modules are discussed in the subsequent sections.

5.3 Experimental Setup

This GA system consists of a Random Number Generator (RNG), a Fitness Module (FM), a Selection Module (SM), a Crossover Module (CM) and a Mutation Module (MM) and these are connected together as shown in figure 5.1 below.

Figure 5.1. Architecture of the GA System Implemented in This Work

The RNG was used to generate the initial population and also to generate random numbers to the Crossover Module (CM), Mutation Module (MM) and the Selection Module (SM). The seed to the RNG is supplied by the user. Once the initial population is generated, the FM evaluates the fitness of each of these members and these fitness values along with the members are passed onto the SM. The Roulette Wheel selection process is used for all implementations. The random numbers supplied by the RNG help determine which members are selected as *parents*. These parent chromosomes are then crossed over

to produce *offspring* which are then mutated. After mutation, the resultant chromosomes are evaluated by the fitness module. Once the new generation is complete, these members replace the old population. Since a total replacement scheme is employed, this system is based on a simple GA system.

The library of modules constitutes different types of RNGs and CMs, while one type of SM and MM is implemented. The modules from table 5.1 were used interchangeably in the architecture shown above [figure 5.1.].

The Verilog HDL and VHDL simulations were carried out on the Cadence platform using NCSIM version 5.10 and SIMVISION was used to view the output waveforms. The Cadence software was run on the UNIX platform on the SUN SOLARIS workstation which was the Ultra10 system. The MATLAB version 7.0.1 was used for the simulation of the GAOT toolbox.

In the first simulation run for the fitness function $f(x) = 2x$, a LFSR based RNG, a Roulette Wheel selection, a One-point crossover and a mutation function were employed. This implementation was then simulated for each of the different Crossover Modules. The next implementation consisted of a Multiple LFSR based RNG, a Roulette Wheel selection, a One-point crossover and mutation module. Again, this setup was repeated for different crossover modules. Next, the same process was repeated for the second fitness function $f(x) = 2x^3 - 45x^2 + 300x$. This entire procedure was carried out for an 8-bit GA

system and a 16-bit GA system. The maximum value obtained by each GA system gives a measure of the performance of the system. The simulation results obtained for the different implementations are given in section 5.5.

5.4 Functional Verification

The GA architecture illustrated in figure 5.1 was simulated to verify correct functionality and to analyze the performance. The performance analyses consisted of a detailed comparison between the results obtained for the different GA system implementations. The GA modules were functionally verified in two steps. The first step involved the functional verification of each module individually, where the modules were tested to operate correctly under all possible conditions. In the second step, the modules were connected together in the configuration given above [figure 5.1] and simulated for the two fitness functions. During these simulations, each module and the system as a whole were checked for functionality. The successful completion of these two steps ensured the proper working of the modules and the entire system.

The maximum value obtained by each GA system gives a measure of the performance of the system. For all the simulations, a population size of 16 was used and the GAs were simulated for 16 runs, each with a different initial seed value for the RNG. Two member sizes of 8 and 16 were used which formed the 8-bit GA system and the 16-bit GA system respectively. The width of the fitness values for the 8-bit system was 12-bits, whereas the fitness values were 32-bits long for the 16-bit GA system. For ease of performance

comparison, the same seed was used for all the 8-bit GA systems where the initial population was generated by the LFSR based RNG and a different set of seeds were used for all systems using the multiple LFSR based RNG. The same method was adopted for the 16-bit implementation. This population set and parameter sizes were used as they provided good coverage of the search space and were sufficient for functional verification of the modules.

The fitness functions were maximized over the domain $0 \leq x \leq 255$ for the 8-bit GA system and $0 \leq x \leq 65535$ for the 16-bit system.

The Verilog HDL and VHDL simulations verified the correct functionality of all the GA modules and systems. Also, an increase in the average fitness value of the GA system was observed when compared to the initial population.

5.5 Performance Analysis

Once the modules and systems were functionally verified, the performance analysis was carried out. The maximum value attained by the system gives a measure of the performance.

For the 8-bit GA system to maximize the function $f(x) = 2x$ using an LFSR based RNG, four implementations exist. They are

- LFSR based RNG using an Arithmetic Crossover module
- LFSR based RNG using Uniform Crossover module
- LFSR based RNG using Two-point Crossover module
- LFSR based RNG using One-point Crossover module

Each of these implementations was simulated for 16 runs, with a unique seed for each run. The maximum fitness value for each of these runs was observed and the maximum, minimum and average values for the set of maxima were tabulated. The standard deviation and the 95% confidence interval were also calculated and all these are tabulated in Table 5.2.

	Maximum	Minimum	Average	Standard Deviation	Confidence Interval
Initial population	460	416	444.625	15.40076	(437.08,452.17)
LFSR with Arithmetic CM	502	438	492.375	17.31618	(483.89,500.96)
LFSR with Uniform CM	496	402	443.25	23.61779	(431.68,454.83)
LFSR with Two-point CM	508	472	497	13.77921	(490.25,503.75)
LFSR with One-point CM	510	426	474.125	29.62403	(459.61,488.64)

Table 5.2. Performance Results for the 8-Bit GA to Optimize the Function $f(x) = 2x$

The graphs for the different systems are given in the figures below; all systems have 16 runs. From the graphs shown below, it is observed that the fitness value for the initial

population generated by the LFSR based RNG is low. The GA system has shown considerably high fitness values for all the implementations except the one using the uniform crossover module. This could be due to the initial population or because of the random number that decides the bits that are copied from both the parents.

Figure 5.2. Maximum Value Obtained for the Initial Population Generated by the LFSR Based RNG (16 Runs)

The fitness function is then maximized for different GA implementations, where the initial population is generated by Multiple LFSR based RNG. The results obtained for the function $f(x) = 2x$ for the different GA systems is given in table 5.3 below. The functions were maximized over the domain $0 \leq x \leq 255$, hence the maximum attainable fitness

value for x is 510. The results table shows that all the GA implementations attained a maximum value of 510. The Uniform Crossover module did not perform as expected in the previous implementation [table 5.2], but in this implementation it has performed as well as the other modules. Also, the average value of the maximum for 16 runs is higher than those obtained with the LFSR implementation and these values are much closer to the maximum attainable value of 510.

	Maximum	Minimum	Average	Standard Deviation	Confidence Interval
Initial population	506	442	479.6	26.01538	(456.8,502.40)
Multiple LFSR with Arithmetic CM	510	474	485.2	14.25482	(472.71,497.7)
Multiple LFSR with Uniform CM	510	440	485.2	29.88645	(459.00,511.4)
Multiple LFSR with Two-point CM	510	498	503.6	4.560702	(499.60,507.6)
Multiple LFSR with One-point CM	510	438	479.6	37.18602	(447.01,512.19)

Table 5.3. Results for the Function $f(X) = 2x$ Using Multiple LFSR Based RNG

The graph shown below in figure 5.3 illustrates the maximum value obtained by the different GA implementations for 16 simulation runs. Here, it is observed that the fitness function $f(x) = 2x$ is maximized efficiently by all the GA modules. The initial population generated by multiple LFSR based RNG has a high fitness value and this value is better than that obtained by the LSFSR based RNG. In this case study, all the GA modules have

attained the maximum possible fitness value. Also, there is not much deviation between the maximum values obtained for the different runs. The entire simulation run for 16 iterations (to obtain the final population to 16) took 74 clock cycles.

**Figure 5.3. Maximum Value Obtained for the Fitness Function $f(x) = 2x$
Implemented Using Multiple LFSR Based RNG**

A comparison of the simulation results from the LFSR based RNG GA implementations and Multiple LFSR based RNG systems is given in figure 5.4. It is observed here that the initial population generated by the system that uses multiple LFSR based RNG has a considerably higher maximum fitness value than the one that uses the LFSR based RNG.

Also, the initial population generated by the RNGs has a lower fitness value than the results obtained after the population is evolved by the GA. It is also observed that the results from the GA systems using Multiple LFSR based RNG are higher than those obtained by the systems using LFSR based RNG for all the cases.

Figure 5.4. Comparison of Results from the LFSR Based GA Implementations and Multiple LFSR Based GA Implementations

Next, the performance analysis is done for the GA implementations used for maximizing the function $f(x) = 2x^3 - 45x^2 + 300x$. For the 8-bit GA system LFSR based RNG, four implementations exist that are similar to the implementations for the first fitness function. They are

- LFSR based RNG using an Arithmetic Crossover module
- LFSR based RNG using Uniform Crossover module
- LFSR based RNG using Two-point Crossover module
- LFSR based RNG using One-point Crossover module

The results obtained for the simulation of these systems are tabulated below (table 5.4). These results also show that the fitness values obtained for the initial population are less than those obtained once this initial population is evolved through the different GA processes. The maximum fitness values obtained by the GA processes are about 23 – 35% higher than that obtained for the initial population. The averages of all the maxima obtained are also higher for all the different GA implementations. The results are also plotted in figure 5.5.

	Maximum	Minimum	Average	Standard Deviation	Confidence Interval
Initial population	22022500	16847717	19644831	2387642	(19929678,24115322)
LFSR with Arithmetic CM	28866757	28866757	28866757	0	(28866757,28866757)
LFSR with Uniform CM	27124452	21433824	23617961	2639683	(24810710,29438194)
LFSR with Two-point CM	29947108	20288128	26011529	3902130	(26526803,33367413)
LFSR with One-point CM	29947108	18120672	24175125	5369298	(25240797,34653419)

Table 5.4. Results for the GA Implementations Using LFSR Based RNG to Maximize the Fitness Function $f(x) = 2x^3 - 45x^2 + 300x$

Figure 5.5. Results for $f(x) = 2x^3 - 45x^2 + 300x$ Using LFSR Based RNG

The fitness function $f(x) = 2x^3 - 45x^2 + 300x$ is then simulated for the different GA implementations. The results for these simulations are given in table 5.5. From the table it is seen that the fitness values obtained for the initial population are less than those obtained for the different GA implementations. For the fitness function $f(x) = 2x^3 - 45x^2 + 300x$ where $0 \leq x \leq 255$, the maximum fitness value that can be attained is 30313125. This value has been attained by three of the four crossover implementations.

	Maximum	Minimum	Average	Standard Deviation	Confidence Interval
Initial population	29584049	19456177	25250640	4139791	(21622020, 28879261)
Multiple LFSR with Arithmetic CM	28866757	28866757	28866757	0	(28866757, 28866757)
Multiple LFSR with Uniform CM	30313125	18382721	25964649	4674096	(21867698, 30061600)
Multiple LFSR with Two-point CM	30313125	28161153	29447642	936644.3	(28626652, 30268632)
Multiple LFSR with One-point CM	30313125	28512500	29301249	781139.9	(28616563, 29985936)

Table 5.5. Performance Analysis for the 8-Bit GA Systems Maximizing the

$$\text{Function } f(x) = 2x^3 - 45x^2 + 300x$$

The results shown in table 5.5 above are plotted in figure 5.6. It is obvious that the maximum fitness values attained for the different GA systems are very close to the maximum attainable value. Also, the initial population generated by Multiple LFSR based RNG has a higher fitness value than that generated by the LFSR based RNG for the particular fitness function.

**Figure 5.6. Results for the Different GA Implementations for $f(x) = 2x^3 - 45x^2 + 300x$
(Using Multiple LFSR Based RNG)**

Figure 5.7 below illustrates the comparison between the different GA implementations used to maximize the fitness function $f(x) = 2x^3 - 45x^2 + 300x$. It is observed that the results are similar to those obtained for the other fitness function $f(x) = 2x$. In figure 5.7, it is seen that the fitness value obtained for the initial population is the minimum value recorded. When the initial population undergoes different GA processes, the fitness value of the resulting system increases considerably. Also, the simulation run for this GA system took 74 clock cycles.

Figure 5.7. Comparison Between LFSR Based RNG Systems and Multiple LFSR Based Systems for the Function $2x^3-45x^2+300x$

5.6 Comparison with Software-based GA

The two fitness functions have also been run on the GOAT toolbox available in the MATLAB environment to enable comparison between the software and hardware implementations. All the parameters for the MATLAB simulations are the same as those used in the hardware implementation; this makes the comparison process easier. The results obtained for the fitness function $f(x) = 2x$ is shown in figure 5.8. The GAOT toolbox attains a maximum value of 510 after the GA run, which is the maximum

possible value for the function. The same value was obtained by the hardware implementation using the multiple LFSR based RNG.

Figure 5.8. MATLAB Simulation Results for $f(x) = 2x$

The results for the second fitness function $f(x) = 2x^3 - 45x^2 + 300x$ are shown in figure 5.9. The MATLAB simulation results in a maximum value of 3.0×10^7 , which is the maximum attainable value for the function. This value is similar to those obtained for the hardware simulation using multiple LFSR based RNG systems.

Figure 5.9. MATLAB Simulation Results for $f(x) = 2x^3 - 45x^2 + 300x$

Chapter 6

Conclusions and Future Work

This thesis work presents the design of a library of modules to enable hardware implementation of Genetic Algorithms, providing a set of modules and an example architecture which other researchers can customize for specific applications. The various modules of the GA system are described in Hardware Description Languages like VHDL and Verilog HDL. This enables reuse and flexibility for the modules and achieves higher speed for the GA than its software counterpart. These hardware modules have also been compared to the software version described in MATLAB.

Each module was verified to be functionally correct through numerous simulations. The modules were also simulated to analyze the performance. After the correct functionality of each module was tested individually, the modules were connected together and the system was functionally verified. The Verilog HDL and VHDL simulations on the Cadence platform confirmed that each of these modules and the different GA systems were functionally correct.

From the performance analysis of the different modules, we conclude the following:

- the initial population generated by multiple LFSR based RNG is more suitable for use with GA systems over the LFSR based RNG. From the simulation results, it was obvious that the fitness values for the population generated by multiple LFSR based RNG were higher than the simple LFSR based RNG
- the type of crossover operator used in the GA system did not have a major effect on the performance of the system except with the Uniform Crossover GA implementation using the LFSR based RNG. This module did not work as expected, probably because of the initial population or the random bits from the two parent chromosomes that formed the offspring. Apart from this, any crossover module discussed in this thesis work can be used to implement an efficient GA system.
- the performance results for the GA system maximizing the function $f(x) = 2x$ and the function $f(x) = 2x^3 - 45x^2 + 300x$ were good. Both the systems were able to find the maximum attainable value for the function in most of the cases. This demonstrates that GAs are efficient systems for optimizing specific functions.
- the number of clock cycles to generate the new population (considering a population size of 16), took 74 clock cycles. Considering a 1.2GHz system, this would produce a delay of 61.42ns. This time delay could be further decreased by using parallelized GA systems, each producing a set of population members.

The modules implemented in this thesis work only deal with binary representations. This work can be extended to include real number representations. Including real number representation will allow the GA system to be used for a wide range of applications.

These floating point implementations can be realized in accordance with the IEEE 754 floating point standard.

The Verilog and VHDL modules implemented in this work can be synthesized to obtain the gate-level netlist and meet constraints like area, power and speed. These modules can also be used by other researchers for specific applications and further mapped to Application Specific Field Arrays and Field Programmable Gate Arrays.

Bibliography

- [1] Goldberg, G.A., *Genetic Algorithms in Search, Optimization and Machine Learning*, Pearson Education, Inc., 2002.
- [2] Palnitkar, S., *Verilog HDL – A Guide to Digital Design and Synthesis*, Pearson Education, Inc., 2004.
- [3] Teodorescu, H., Jain, L.C., and Kandel, A., *Hardware Implementation of Intelligent Systems*, Physica-Verlag, 2001.
- [4] Mazumder, P. and Rudnick, E.M., *Genetic Algorithms for VLSI Design, Layout and Test Automation*, Prentice Hall, 1999.
- [5] Graham, P. and Nelson, B.E., “A Hardware Genetic Algorithm for the Traveling Salesman Problem on SPLASH 2”, *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, pp. 352 – 361, 1995.
- [6] Scott, S. D., “HGA: A Hardware-based Genetic Algorithm”, Master’s Thesis, University of Nebraska, 1994.
- [7] Shackleford, B., Okushi, E., Yasuda, M., Koizumi, H., Seo, K., Iwamoto, T., and Yasuura, H., “High-Performance Hardware Design and Implementation of Genetic Algorithms”, in Teodorescu et al, *Hardware Implementation of Intelligent Systems*, pp. 53 - 87, 2001.

- [8] Koonar, G.G., “A Reconfigurable Hardware Implementation of Genetic Algorithms for VLSI CAD Design”, Master’s Thesis, University of Guelph, 2003.
- [9] Apornetewan, C. and Chongstitvatana, P., “A Hardware Implementation of the Compact Genetic Algorithm”, *Proceedings of the 2001 Congress on Evolutionary Computation*, pp. 624 - 629, 2001.
- [10] Graham, P. and Nelson, B.E., “Genetic Algorithms in Software and Hardware – A Performance Analysis of Workstation and Custom Machine Implementation”, *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 216 – 225, 1996.
- [11] Tommiska, M. and Vuori, J., “Hardware Implementation of GA”, *Proceedings of the Second Nordic Workshop on Genetic Algorithms and their Applications (2NWGA)*, 1996.
- [12] http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999f/Drivers_Ed/lfsr.htm
l viewed on 03/25/05.
- [13] http://www.maxim-ic.com/appnotes.cfm/appnote_number/1743 viewed on 12/04/04.
- [14] Miller, B.L. and Goldberg, D.E., Genetic Algorithms, Tournament Selection and the effects of Noise, Technical Report, University of Illinois, 1995.
- [15] <http://www.econ.iastate.edu/tesfatsi/holland.GAIntro.htm> viewed on 04/05/2005.
- [16] http://klabs.org/richcontent/Tutorial/MiniCourses/fsm_mapld_2002/24_lfsr.ppt
viewed on 04/23/05.
- [17] <http://www.csit.fsu.edu/~burkardt/fsu/poster.ppt> viewed on 04/25/05.
- [18] [http://courses.washington.edu/tcss588/388,8,Slide 8](http://courses.washington.edu/tcss588/388,8,Slide%208) viewed on 02/20/05.

- [19] Houck, C., Joines, J., and Kay, M., A Genetic Algorithm for Function Optimization: A Matlab Implementation, NSCU-IE Technical Report, 1995.
- [20] Martin, P., An Analysis of Random Number Generators for a Hardware Implementation of Genetic Programming using FPGAs and Handel-C, Technical Report, University of Essex, 2002.
- [21] Knuth, D.E., Seminumerical Algorithms, volume 3. Addison-Wesley Publishing Company, 1969.
- [22] Marsaglia, G. Web Site for Diehard Random Number Test Suite. <http://stat.fsu.edu/geo/>, 2001 viewed on 15/06/2001.
- [23] Meysenburg, M.M., and Foster, J.A., "Random Generator Quality and GP Performance", *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pp 1121–1126, Orlando, Florida, USA, 13-17 July 1999.
- [24] Chamber, L.D., Complex Coding Systems, *Practical Handbook of Genetic Algorithms*, volume 3, pp 65-85, CRC Press, 1999.
- [25] <http://www.stephenwolfram.com/publications/articles/ca/> viewed on 12/10/04.
- [26] Cantu-Paz, E., "On random numbers and the performance of genetic algorithms", *Proceedings of the Genetic and Evolutionary Computation Conference*, pp 311-318, Morgan Kaufmann Publishers, San Francisco, 2002.
- [27] <http://cs.felk.cvut.cz/~xobitko/ga/selection.html> viewed on 11/10/04.