GENERAL PURPOSE EVOLUTIONARY ALGORITHM TESTBED

A Thesis

presented to

the Faculty of the Graduate School

at the University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

KIRAN KUMAR TATI

Dr. Tina Smilkstein, Thesis Supervisor

DECEMBER 2009

The undersigned, appointed by the Dean of the Graduate School, have examined the thesis entitled

GENERAL PURPOSE EVOLUTIONARY ALGORITHM TESTBED

presented by **Kiran Kumar Tati**

a candidate for the degree of

Master of Science

and hereby certify that in their opinion it is worthy of acceptance.

Dr. Tina Smilkstein, Assistant Professor, Department of Electrical and Computer Engineering

---

Dr. Naz Islam, Associate Professor, Department of electrical and Computer Engineering

---

John Fresen, Assistant Teaching Professor, Department of Statistics

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank each and everyone who directly or indirectly influenced me to realize this dream of mine. I wish to express my heart-felt gratitude to my graduate advisor, Dr. Tina Smilkstein for her valuable guidance, support, patience and invaluable suggestions during the entire course of this work.

I would also like to thank Dr. John Fresen and Dr. Naz Islam for readily accepting to be in my thesis examining committee and providing help and valuable suggestions on my thesis report.

I would like to take this opportunity to thank my friend Gayatri Kallepalle and my research mates for their help and encouragement at all stages of my thesis.

I express my gratitude to the entire faculty and staff of Department of Electrical and Computer Engineering, who cooperated a lot during my course of study at Mizzou.

I take this opportunity to express my gratitude to my mother Vara Lakshmi, my father Jagan Mohan Rao and my brother Vinod Kumar for their love, care and support. I thank all my friends and well wishers who stood by my side, helped me sail during my tough times and provided valuable suggestions throughout my research work.

Finally, I wish to think of the Universal Being for all that He has given me.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figures.....Page

GENERAL PURPOSE EVOLUTIONARY ALGORITHM TESTBED

Kiran Kumar Tati

Dr. Tina Smilkstein, Advisor

ABSTRACT

We developed a "General Purpose Evolutionary Algorithm Testbed" through which a wide variety of evolutionary algorithms can be implemented in hardware, quickly and with a little hardware knowledge. A user interface allows the user to enter parameters needed to choose evolutionary algorithm components such as type of encoding, initial population description, selection type, reproduction type and the type of evaluation and termination criteria. The testbed is programmed based on those parameters and allows the user to make a variety of sensor connections and computer connections through which intrinsic and extrinsic evolutions and debugging are made possible.

Our system provides a computer based graphical user interface with different windows showing all possible options for each system description parameter. VHDL code is then generated according to the inputted parameters which implements an FSM which controls the flow of data through the evolutionary algorithm system. The state machine performs all the steps of evolutionary algorithm. The design is run on Xilinx ISE

using ModelSim XE-III (MXE-III) simulator and Xilinx® Spartan 3E™ starter kit is used as testbed. . A breadboard will be added in future versions to make interfacing to external sensors and test systems less arduous. A RS-232 connection is used to connect the testbed back to computer for debugging purposes. Designers require minimum hardware knowledge to use our graphical user interface, program the testbed, connect the sensors and debugger. With this system we tried to decrease the barrier of evolutionary algorithm designers to implement their designs in hardware and allow for easier debugging, revision and research on evolutionary hardware systems.

In this thesis we also evaluate the resources used for various evolutionary algorithm systems. We used subtractive model of color theory as an application where tertiary colors evolve from three basic colors red, yellow and blue. We observed solutions moving towards global maxima instead of looking for local maxima with introduction of randomness. We found that the type of reproduction had an effect on the resources used whereas the change in other parameters such as population, iterations, termination criteria and debug shows normal behavior.

CHAPTER 1

INTRODUCTION

1.1 Problem

Evolutionary algorithms (EA) are algorithms that may be used to solve real world problems by imitating the "survival of the fittest" model encountered in nature. It is a method that is different than, say, artificial intelligence production system models in that EAs do not require a full set of rules on how to find a result before they can run. Their strength comes from this and the fact that they keep a certain amount of randomness among the possible solutions, a trait that can allow them to find solutions to problems not envisioned at the time of deployment of the system. Mathematicians and computer scientists have been exploring the possibilities of EAs for many years [1-3]. For mathematicians and computer scientists, moving their ideas into hardware may be too much of a challenge due to the required need of hardware knowledge. Even if they do rise to the occasion and build a system in hardware, each change is often painful and time consuming. Once the circuit is built, debugging and performance analysis is needed. With dedicated systems each time a new system is built a new way to debug and evaluate must also be developed. All of this has provided a barrier for those without hardware skills and knowledge and often keeps their work in the software realm. In order to enable those, we

have designed a reprogrammable EA algorithm testbed for quick implementation of EAs in hardware.

Because there are no substrates available made especially for EA systems some type of reconfigurable device must be used. FPGAs and microprocessors are common substrates for the EA hardware systems found in published papers but, because those substrates are not specifically for the development of EA systems their user interface, debugging interface and design entry method are not natural to the development and debugging of an EA system. In the design of our general purpose EA testbed, "GPeat", not only was design effort put into the development of hardware structures to enable efficient execution of evolutionary algorithms, much effort was also put in to making the user interface specific to the development of EA systems. In this way we developed GPeat which can be used to design and run a wide variety of evolutionary algorithms as well as allowing a person with minimal hardware knowledge to design EA systems in hardware.

1.2 Hardware Platform

Evolutionary algorithms can be implemented using platforms such as programmable DSPs, FPGA's and general purpose processors. Programmable DSPs are good for applications that use many multipliers, adders and other mathematical structures but do not provide structures that support, for example, sorting, crossovers and mutations and other common procedures used in EAs. They also consume more power and are not flexible enough to support applications that stray from patterns that signal processing applications require. General purpose processors process data in series. They are easy to

program but, because they cannot run calculations in parallel they cannot give the speed improvements that an ASIC (application specific integrated circuit) can. ASICs can provide the most efficient and fast circuitry for EAs but they are usually specific to a single problem and are dedicated to give the best results only for that problem. If a new EA application was to be designed, a new ASIC would have to be designed.

Our GPeat system has been implemented on a Field Programmable gate array (FPGA). EAs need certain features in hardware to make them fast and to make them use resources efficiently. The flexibility and reconfigurable feature of FPGAs make them a good platform for implementing GPeat. FPGAs allow for each new EA system implemented on GPeat to be optimized for speed and memory usage. The hardware structures that allow a speed up of slow data manipulations by using parallel processing can be reconfigured to fit each specific application. Memory usage can be customized to fit the EAs population and iteration needs. And, because of information provided during compilation of designs for an FPGA, resource use can be accurately and quickly analyzed. FPGAs also have the advantage of fast time-to-market and, recently, high capacity. These features made an FPGA the most advantageous programmable platform for implementing GPeat.

1.3 Report Organization

The basic theory and an introduction of evolutionary algorithms along with a brief review of existing EA hardware systems are presented in Chapter 2. In Chapter 3 the general purpose evolutionary algorithm testbed system (GPeat) description is reported. The VHDL description for the implementation is given in Chapter 4. ASM charts for

description are also presented in chapter 4. Simulation and synthesis results are reported in Chapter 5. Hardware resources usage and performance are reported along with a discussion of results are included with the simulation and synthesis results. Conclusion and future work is reported in Chapter 6.

CHAPTER 2

BASIC THEORY AND REVIEW

In this chapter the basic theory and history of evolutionary algorithms are presented as well as an overview of evolutionary algorithm systems.

2.1 Evolutionary Algorithms (EA)

Evolutionary Algorithms are search algorithms inspired from natural evolution. These algorithms maintain a population of individuals where each individual is a possible solution to particular problem that is encoded in some fashion. Initially, the population of individuals are randomly generated or generated according to some set of rules. Genetic operations such as selection, reproduction (crossover and mutation) are applied to generate new populations. Fitness, a numeric value that measures the quality of that individual, is calculated according to some objective function. Some individuals may be good in some ways and be bad for that particular problem in other ways. The best ones (based on the selection criteria) from the population are taken back to generate a new population of individuals. Members of the new population go through the same process; they reproduce, are evaluated and then, in most cases, the better (stronger) members of a population are selected to continue on into the next generation. The process will continue until the termination criterion is achieved. The basic structure of EA is shown in fig 2.1.

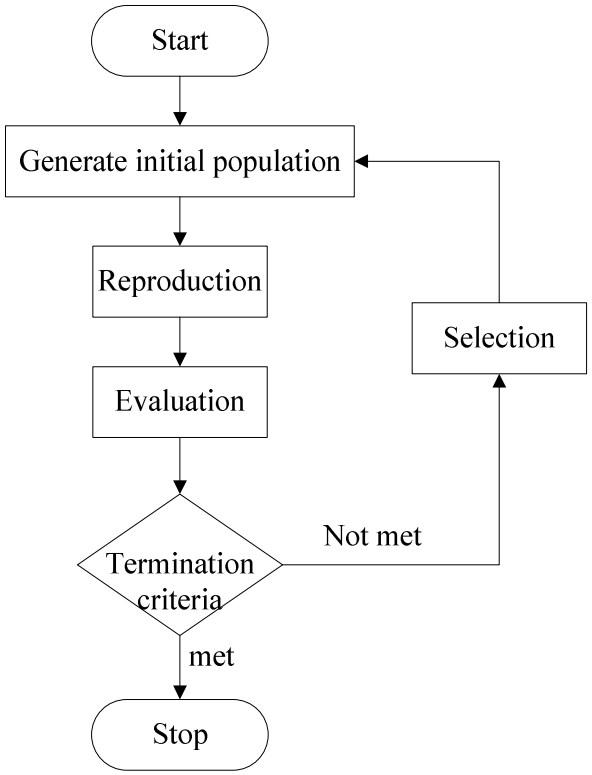

Figure 2.1 Structure of EA

2.1.1 Encoding (Representation)

Each individual has information that can be used as a solution and is encoded into a form so as to be suitable for application of genetic operations. An individual is represented as a "chromosome" and has all the problem parameters or values that need to be present in a solution of a particular problem. In EAs these problem parameters or units that make up a chromosome are called "genes". The most commonly used encodings are binary strings, integers, real numbers, graphs, hybrids.

Gene 1	Gene 2	Gene 3	….	…..	…….

Chromosome

Figure 2.2 Chromosome structure.

In binary string encoding, each individual is represented in sequences of 1's and 0's. A single bit or a group bits form a gene, where each gene represents some aspect of solution. In turn, a group of genes makes up a chromosome. Similarly an array of integers for integer type, a set of vertices for graphs, an array of real numbers for real number type and some component values (like switch positions in circuit topologies) for hybrids form a gene.

For example there might be the case of an autonomous car where each chromosome is encoded in the form of four genes of binary strings of length two bits. Each gene represents a different aspect of the solution. For example, the first gene might represent the direction to move (such as driving forward, backwards, left, right); the second gene might represent the speed, etc. Other encodings follow this same pattern of encoding different information in separate genes and combining them into a single chromosome.

2.1.2 Selection

Selection is a process of taking individuals from one generation to be used in forming the next population. There are a large number of selection methods. Some commonly used selection types are elitism, roulette-wheel, tournament and generational selection. Some of these types may not provide the required number of good individuals to form next generation. In that case, the remaining individuals of generation are either generated randomly or according to some set of rules. The randomness provided by introducing some new individuals to a population help keep the population from being concentrated around a local solution when there may be a better, global solution

available. Randomness allows a more complete search of the solution space. The higher the number of individuals provided by the selection type the less random a population will be and the more time it will take the algorithm to find a global solution that is away from the present population. This ability to search a solution space allows EAs to find solutions that a rule-based system may not be able to find and solve problems that a system designer might never have envisioned during the development of the system.

In elitism selection, some percentage or number of the best individuals are taken from one generation in order to form the next generation. That percentage is called as elitism rate.

Roulette-wheel selection derives its concept from the roulette game where an individual's space on the wheel is based on its fitness value i.e., random selection is done with chance of picking an individual is proportional to its fitness.

In tournament selection, each population is sub-divided into groups and a single best individual is taken from each group to form new population.

In generational selection, all the individuals from previous generation are taken to be the next generation.

2.1.3 Reproduction

Reproduction produces off-spring (new individuals) from parents (existing individuals) by applying genetic operators. The most common genetic operators used for reproduction are crossover and mutation.

2.1.3.1 Crossover

In crossover, genes of two or more parents are exchanged to create off-spring. The off-spring inherit some of the qualities of each of the parents resulting in a change in their fitness. Better individuals are not necessarily formed by taking best characteristics from their parents but assuming a solution space with local and global maxima, an individual with a better fitness will likely have some traits that locates it near one of these regions. Taking pieces of a chromosome that is near a maxima will possibly create a child that has inherited some of the traits that located the parent near a maxima. As new generations are created using crossovers, traits that locate individuals near a maxima are concentrated and chromosome quality may be improved. A problem occurs when the chromosomes approach a local maxima that is not good enough satisfy the system users' minimum acceptable fitness quality. The randomness introduced by doing crossovers can help to make sure that the individuals in a population do not get stuck on a local maxima by "knocking them off" a maxima. Crossover probability decides the occurrence of crossover during evolution. Some of the existing crossover techniques are single-point crossover, two-point crossover, uniform crossover, arithmetic crossover etc. These types are explained in detail in section 3.2.1.2.

2.1.3.2 Mutation

Mutation is a genetic operator where off-spring are created by the changing of one or more genes in a single parent. Here mutation probability controls the rate of occurrences of mutation during evolution. Some of the existing mutation techniques are

flip-bit mutation, exchange adjacent gene mutation, mirror mutation etc. These types are explained in detail in chapter 3.2.1.2.

2.1.4 Evaluation

Evaluation is a process where an "objective function" is used on each individual in order to assign some fitness value (a numeric score) which indicates the quality of that individual as a solution to a particular problem.

Fitness values that are turned into percentages will represent how close an individual is to ideal individual. Generally, an objective function maps an individual's fitness onto a real number scale. Selection types make use of fitness value for selecting chromosomes for future generations.

2.1.5 Termination Criteria

Based on the termination criteria or stopping criteria, the process of evolution is stopped. Termination criteria can be based on the number of generations processed or fitness or both.

In generations based termination criteria, the process of evolution will stop after running for specified number of generations, irrespective of the fitness value. With this type, we can limit the time for running the process but there is a chance of ending up with no individual of satisfactory fitness. Sometimes the process will continue for all specified generations even if the best individual is found in the middle of the run. This results in the wasting of time by continuing the search even though a suitable solution has already been found.

In fitness based termination criteria, process will continue until an individual of specified fitness is found irrespective of number of generations [4]. An individual with fitness of choice can be achieved but there is no control on time taken for running the process. If only this criteria is used there is a possibility that the search will never end. If a solution of the fitness required is never found the system will continue to search forever.

By mixing above two types an intermediate termination criteria type can be formed. In this type, the process may continue for all specified generations or, if an individual with the required fitness is found then the system will stop. With this type, time wasting and the infinite time loop problems are solved but a satisfactory individual is not always guaranteed.

The flow and termination criteria of both GPeat and EA systems in general can be better understood by referencing Fig 2.1.

2.2 Additional Comments

Evaluation processes can be done intrinsically (in a real environment) or extrinsically (in a simulator). Extrinsic and intrinsic evolutions each have their own advantages and disadvantages. Most of today's circuit syntheses are done on simulators. For larger systems intrinsic evolution is preferred to reduce their execution time [4].

2.3 Present Systems

Modern world artificial intelligence applications need flexibility in problem solving which can be complicated to achieve using their traditional machine learning and

optimization methods. EAs are inspired by natural evolution and can be used to solve problems where a complete set of rules cannot be generated. In cases where a complete rule set cannot be generated or in cases where problems are computationally expensive, EAs can have an advantage over traditional production system type designs (IF-THEN-ELSE rule based systems).

At first application of EAs was theoretical and limited [5], [6]. Later, developments in the field helped in extending their applications to abstract mathematical problems like bin-packing, a technique where n items from different categories must be put in bins of the same size in a way to minimize the number of bins required. Other problems where EA methods have been applied include structural optimization, pattern recognition, classification etc. [7]. Rapid growth of computational power and the availability of internet allowed scientists to apply their EA theories to many real world areas like robotics, engineering, automotive design, biomimetics, optimized telecommunications routing, computer games, finance and investment strategies and marketing etc. Scheduling problems fit well with EA abilities and systems for such things as airport scheduling, travelling salesman problems, stock market prediction, and portfolio planning, texture analysis have also been targets for the application of EAs [8-18].

Scientists explored EAs in many ways. EAs have immense computational power because of their iterative and random nature. These features of EAs must explore to maximum extent so as to have better results. Hardware implementation of EAs is advantageous with present electronics [18], [19].

EA systems can be implemented on different commercial platforms which include programmable analog platforms, programmable digital platform, general purpose processors and some other research based and commercially available platforms. But each platform has its own advantages and disadvantages [20-22] and [4].

Programmable DSPs can be used for implementing EA systems but they are optimized for signal processing applications and offers blocks such as multipliers and adders, not structures that are specifically useful in implementing EAs. They consume more power and not flexible [23] and [24].

General purpose processor run their programs in series and is not able to exploit parallelism as purely hardware implementations of algorithms can.

Application specified integrated circuits (ASICs) can also be used for implementing EA systems. Our ultimate goal is to build a custom circuit that has specific reconfigurable structures that enable efficient EA system design. This chip would be similar to FPGAs which have specific hardware structures, LUTs, to enable efficient implementation of digital designs and programmable DSPs which have specific hardware structures, multipliers and adders, to enable efficient implementation of DSP algorithms. But as of now, the documented EA systems on ASICs can only do a signal, specific EA and cannot be reconfigured. They are dedicated and are not good for running different EAs. Platforms developed for research purposes are also specified to a particular EA [8-18].

Programmable devices generally come with user interfaces but those interfaces are for general purpose use and are not specific to EAs. As such, designing EAs through

them requires knowledge of not only EA systems it requires knowledge of the underlying hardware and that hardware's software. This is another barrier to the design of efficient physical systems for the engineers that understand software and software implementations of EAs but do not have the knowledge to move their systems to hardware.

As mentioned, each platform has its weaknesses. That disadvantage may be a lack of flexibility or sequential processing which means the advantages of hardware have been wasted. Many have costs in terms of tradeoffs. For example flexibility may be increased with a higher priced platform or speed may be given up to reduce power. None of these are made specifically for EAs and therefore all have overhead and are inefficient.

2.4 Proposed System Justification

With this examination and from basic theory, we determined that a general purpose evolutionary algorithm testbed with a simple GUI can be built that would aid those with weak hardware backgrounds to move their designs into hardware (and therefore real the world realm) would not only be useful but it would aid in the advancement of EA research and products.

The ASIC EA systems and the systems implemented of configurable devices covered a wide array of applications but all of them follow a basic sequence of tasks. After an examination of 40 to 50 documented EA systems and a survey of textbooks on the subject we determined that there is a common flow as shown in Fig 2.3 [25]. From this we were able to identify the blocks which become the bottleneck and restrict performance and develop specific structures to speed up or optimize for EAs.

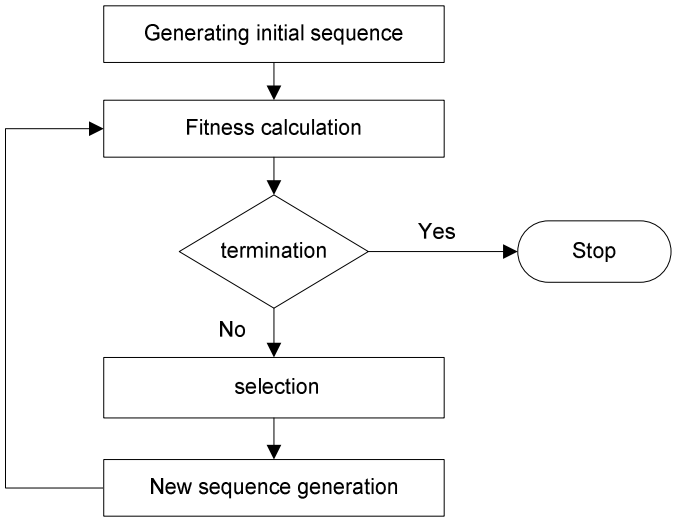

Figure 2.3 Simplified EA hardware flow

Each block of Fig2.3 (which shows documented EA's flow) has a specified function satisfying certain application. Sequence from initial sequence generation block is either generated randomly or according to some set of rules. Fitness calculation block supports only intrinsic or extrinsic evaluation. Selection type is also fixed and new sequence generation block uses fixed types of genetic operators while a wide variety are available. To extend these systems for running multiple EAs each and every block shown in Fig 2.3 must be generalized. This generalization is achieved by providing all available blends (at least most commonly used) of that particular block i.e. fitness block should support both intrinsic and extrinsic evaluation, most commonly used genetic operators should be available. Our system provides all the necessary features needed for running multiple EAs (explained in detail in chapter 3).

A hardware platform with required features for generalizing blocks of Fig 2.3 is needed. With specified hardware structures, adders, multipliers, FPGAs can be used to generalize each and every block overcoming disadvantages of different platforms.

FPGAs have advantage of high time-to-market and low cost while providing good programmability/reconfigurability feature.

A good user interface is also needed for general purpose system. An EA system has to respond to changes in its environment leading to user's interaction with hardware. Our system provides a friendly graphical interface through which user can communicate well with the system.

With these features our system can run multiple EAs while overcoming the above mentioned disadvantages of different systems.

CHAPTER 3

SYSTEM DESCRIPTION

3.1 Introduction

In this chapter a General Purpose Evolutionary Algorithm Testbed (GPeat) system is described. A user interface and hardware form a GPeat system. The hardware is programmed with the EA parameters set by the user through the user interface. Thus programmed hardware runs independently of the user interface and the testbed stops running as the hardware meets user specified termination condition or on a system reset is done.

This chapter also discusses how our system can support a variety of EAs.

3.2 System Description

A GPeat system includes a computer based Graphical User Interface (GUI) and Field-Programmable Gate Array (FPGA) based hardware. The user can set EA parameters through GUI and, based on those values, the hardware is programmed. Once the hardware is programmed the GPeat system is ready to use. As described in section 2.2, the fitness value calculations for the EA can be done either intrinsically or extrinsically. In this version of GPeat, if the user has selected an extrinsic run, the user

must provide the necessary simulator or software and, if an interface to the computer other than through an RS232 connection is used, has to take care of the communication with the board. For example, SPICE, a well known electronic simulator which runs on a can be used as an external simulator. If the simulation is run on another device, the interface to whatever that device is must be created by the user. The high level GPeat system structure for an extrinsic run where the simulation is run on a computer is shown in figure 3.1.

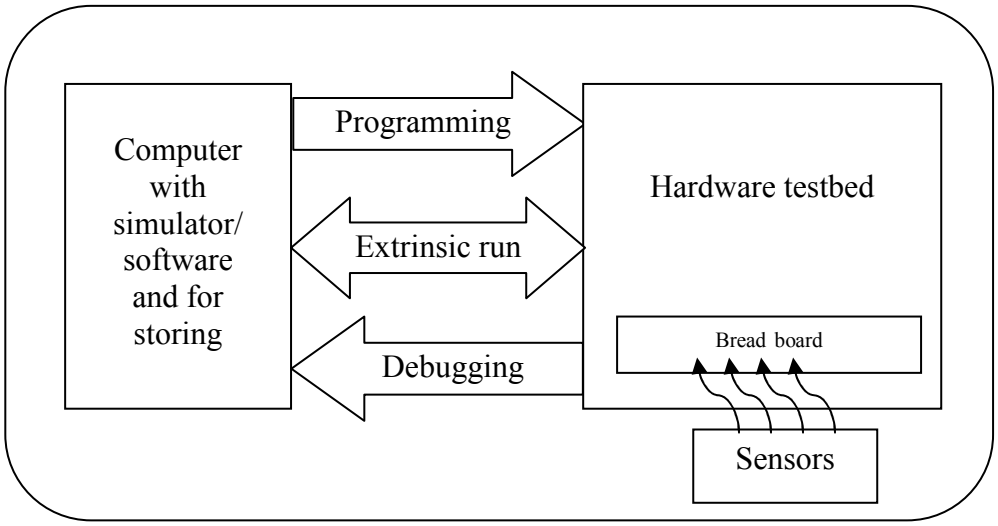

Figure 3.1 High level GPeat System Structure

In the case of intrinsic runs, the fitness is evaluated in a real environment. The GPeat system will, in future versions, have a bank of sensors that can be connected to sensors in the environment. In intrinsic systems that need to gauge the fitness of a chromosome through changes in the environment, these sensors banks will allow a direct connection to GPeat. GPeat can be programmed to evaluate a sensor reading according to rules or take the input of the sensors as the fitness value itself. An example of a system that could use rules to evaluate a sensor reading and give a fitness value would be an

autonomous car. If a chromosome makes the car run into a wall, all chromosomes that drive the car forward could be given a low fitness. An example where the sensor value is the fitness value itself is the signal to a voltage controlled oscillator. The feedback generally describes how far out of synchronization the generated clock is for the target phase so the inverse of that feedback signal would be a meaningful fitness value possibility

The debugging interface provides the feedback which helps the user analyze the performance of their system. Through a RS232 connection or USB connection chromosomes and their corresponding fitness values are sent back to the computer and stored into a log file along with the population size, iteration number, termination criteria, crossover and mutation types along with their sub type, elitism rate, run type, fitness evaluation type and gene type as header. The values are returned for the user to analyze and make decisions about the values of their parameters. . For example, the performance when a particular crossover type and fitness value have been specified data can be stored and later recalled to compare it to the performance of another type of crossover type. This will help the user to know which crossover type is suited better for a particular application.

3.2.1 Graphical User Interface (GUI)

GUI provides graphical windows so the user through can set all the six groups of EA parameters needed to program the board. Those six groups contain information regarding initial population generation, crossover and mutation control, system control, output control, fitness function and sensors. Some of these parameters are mandatory

while others are optional. The details of dependent values and required values are summarized in a table after the description of the values that may be set. A screen shot of GUI is shown in Fig 3.2.

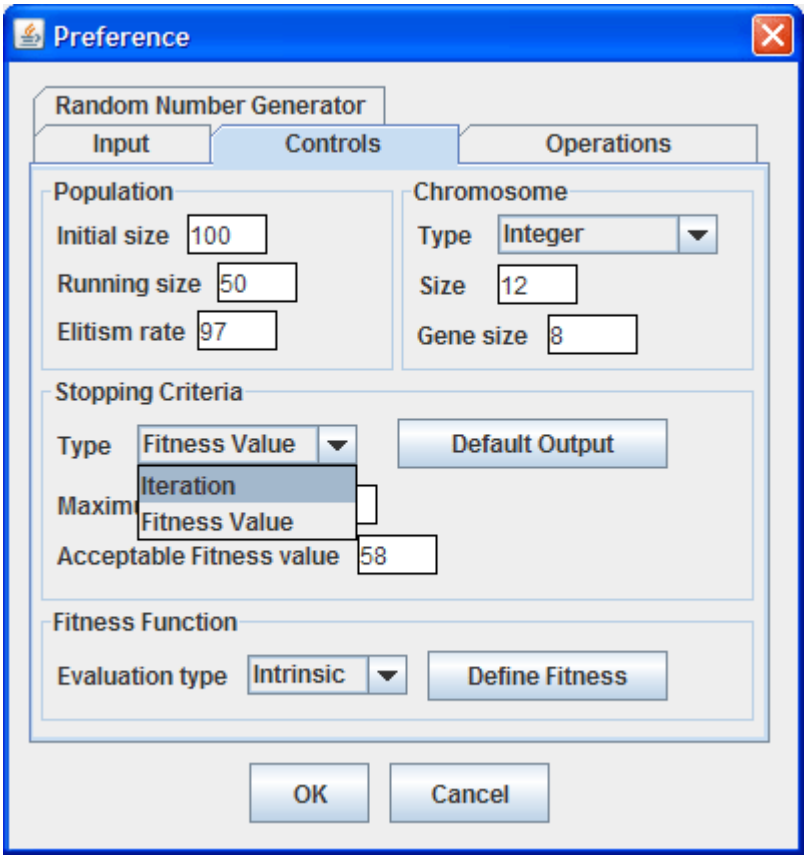

Figure 3.2 Screen shot of GUI

3.2.1.1 *Initial Population Generation Group*

Initial Population Generation Group parameters are population size, chromosome size, gene size, gene type, seed, increment, multiplier and modulus. Population size is the size of initial population. Gene size is the size of each gene and chromosome size is the number of genes of user specified gene type. Gene type defines the type of encoding (for this version, it can be binary or integer). Seed, increment, multiplier and modulus are

used in generating chromosomes through the use of a random number generator. The type of random number generator we have used is a Linear Congruential Generator, a pseudo-random number generator which is capable of generating long runs. It uses the reoccurrence formula shown in equation 3.1 [26].

$$X_{n+1} = (mX_n + i) \bmod M \tag{3.1}$$

where X_0 is a seed, m is the multiplier, i is increment and M is modulus. Parameters like elitism rate also contribute to initial population generation, even though it is not listed among the initial population generation parameters. The number of chromosomes to be generated by the random number generator is population size for first iteration, from the second iteration on it will be the population size minus the elitism rate.

Values from system parts such as sensors will say whether a particular solution is legal or not. For previously mention autonomous car example, at a particular situation, there may be a wall on right side of the car then chromosomes representing a right turn become illegal, where information about presence of wall on right side is provided by sensors. So the legibility of a chromosome changes as the external environment keeps changing.

3.2.1.2 *Crossover and Mutation Group*

Crossover and mutation group parameters are operation type, crossover type, mutation type, crossover probability, mutation probability, arithmetic type, crossover starting point and crossover ending point. Using these parameters crossover and mutation block (Fig. 3.14) will do the reproduction process. User must choose one of the following methods through the select operation type window to generate new members of a

population: crossover-only, mutation-only, both (means both crossover and mutation), exclusive (means either crossover or mutation but not both) and none (means no operation). There are five types of crossovers (single-point crossover, two-point crossover, uniform cross over, arithmetic crossover and random exchange gene crossover) and five types of mutations (flip-bit mutation, exchange adjacent bit mutation, mirror mutation, flip randomly selected gene) where some of these are selected for reproduction depending on the crossover and mutation value. Crossover probability and mutation probability will decide the occurrence of those selected reproduction types. Arithmetic type, crossover starting point and crossover ending point are sub types of crossover which are needed only for certain operations. The crossover and mutation types are described below.

3.2.1.2.1 *Crossover Types*

GPeat provides five crossover types. They are:

- ***Single point crossover*** – This crossover method can be used for both binary and integer genes. One crossover point is selected and the genes from beginning of chromosome to the crossover point are copied from one parent to the offspring and the rest is copied from the second parent to the offspring. Single point crossover is shown in Fig 3.3.

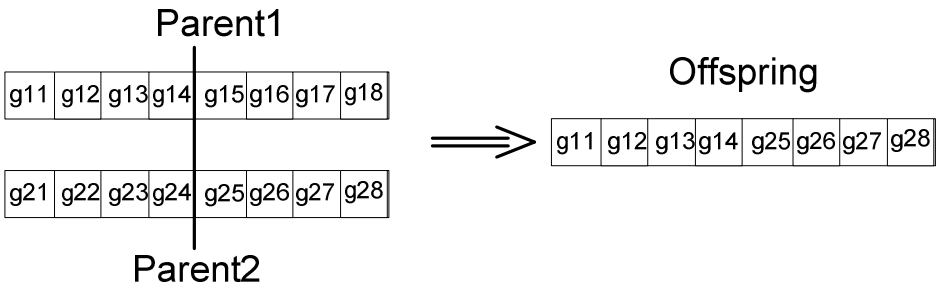

Figure 3.3 Single Point Crossover

where g11,g12,g13,g14,g15,g16,g17 and g18 are genes in parent1, and g21, g22, g23, g24, g25, g26, g27 and g28 are genes in parent2 and the crossover point is shown with a straight line in the figure.

- ***Two point crossover*** - This crossover method can be used for both binary and integer genes. Two crossover points are selected and the genes from beginning of chromosome to the first crossover point are copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent to the offspring. Two point crossover is shown in Fig 3.4.

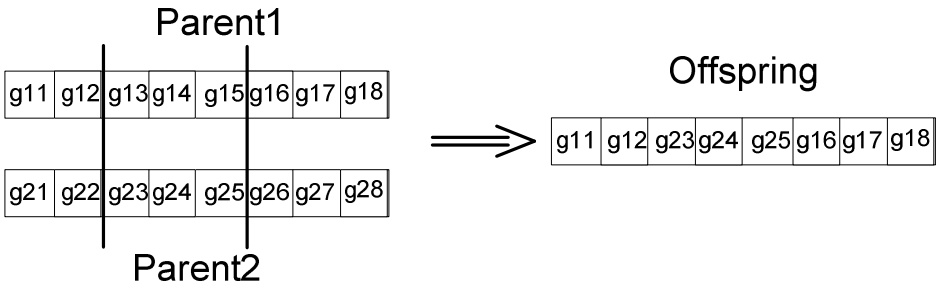

Figure 3.4 Two point crossover

- ***Uniform crossover*** - This crossover method can be used for both binary and integer genes. Genes are uniformly selected from the first and second chromosome i.e., odd genes are taken from parent1 and even genes are taken from

parent2 to form offspring. As they are uniformly selected, user doesn't have to specify any other values. Uniform crossover is shown in Fig 3.5.

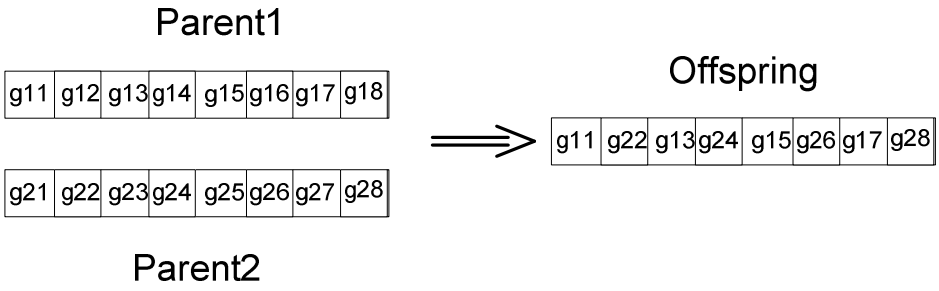

Figure 3.5 Uniform crossover

- ***Arithmetic crossover*** - This crossover method can only be used for binary genes. An arithmetic or logical operation (AND, OR, NAND etc) is performed on two parent chromosomes to make a new offspring. As an example AND type of arithmetic crossover is shown Fig 3.6.

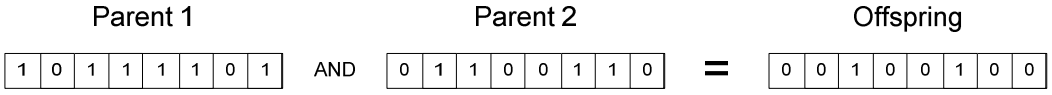

Figure 3.6 Arithmetic crossover

- ***Random shift genes crossover*** – This crossover method can be used for both binary and integer genes. Here the user will specify the number of genes to be shifted. Gene positions are randomly selected and those genes are shifted between the two parent chromosomes to form a new off spring. This crossover is shown in Fig 3.7. In figure three gene positions are randomly selected from parent2 which are shown with an arrow, those genes are placed in gene1 at same positions to form an offspring.

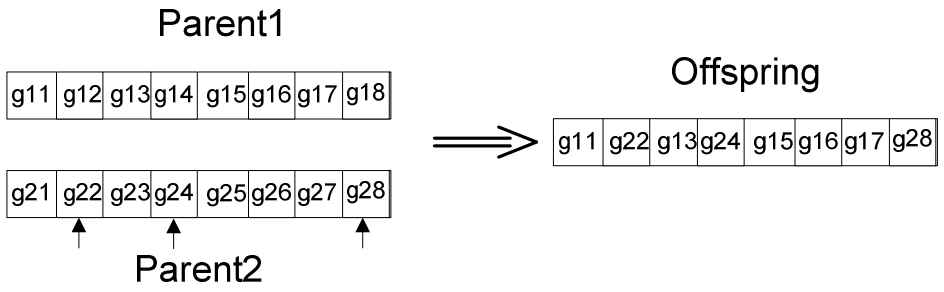

Figure 3.7 Random shift genes crossover

3.2.1.2.2 *Mutation Types*

GPeat provides five mutation types. They are:

- ***Flip bit mutation*** – This mutation method can only be used for binary genes. This method simply inverts the value of each bit in all genes (0 goes to 1 and 1 goes to 0). This mutation is shown in Fig 3.8.

Figure 3.8 Flip bit mutation

- ***Exchange adjacent genes*** – This mutation method can be used for both binary and integer genes. This will exchange adjacent genes of the chromosome. This mutation is shown in Fig 3.9.

Figure 3.9 Exchange adjacent genes.

- ***Mirror mutation*** **–** This mutation method can be used for both binary and integer genes. This will exchange the first half of the chromosome with the second half. Mirror mutation is shown in Fig 3.10.

Figure 3.10 Mirror mutation

- ***Flip a randomly selected bit mutation*** **–** This mutation method can only be used for binary genes. This will flip (0 goes to 1 and 1 goes to 0) a randomly selected bit. This mutation is shown in Fig 3.11. In the figure the randomly selected bit is shown with an arrow.

Figure 3.11 Flip randomly selected bit mutation.

- ***Reciprocal exchange mutation*** **-** This mutation method can be used for both binary and integer genes. This is a special case of reciprocal exchange (swap genes) mutation. Reciprocal exchange mutation picks two random genes somewhere in the chromosome and switches them. This mutation is shown in Fig 3.12. In this figure randomly selected positions are shown with arrows.

Figure 3.12 Reciprocal exchange mutation.

3.2.1.3 *System Control Group*

The system control group parameters are elitism rate, termination criteria, iteration number, acceptable fitness, debug and run type. Generally these parameters decide the system behavior. Elitism rate is the percentage of individuals (chromosomes) from the previous generation that should be added to a new population. There are two types of termination criteria; the termination criteria based on a fitness value stops the system when a chromosome meets user specified fitness ("acceptable fitness" criteria). The other is based on a number of iterations. The system will carry out a number of generations specified by the user and the best of the solutions for all generations are selected at the end. In first type of termination criteria, the user will also be provided with an option to keep going even if a acceptable chromosome has been found until the specified iterations have occurred and then the best chromosome found. For that number of iterations will also be available from the system. For example, if the user defined acceptable value is 90 and the iteration number is 10. For the first type of termination criteria, the system will run until a chromosome of fitness value 90 or above is found (it will run for all iterations if the user wants it to). For the second type, the system will just run for 10 iterations irrespective of fitness value.

The debug option an extra block in the GPeat system which taps values and returns them to the computer. It stores the information of a run in a computer log file with

parameters as header. The information includes chromosome, fitness value, type of crossover and mutation, iteration number and population size, stopping criteria, elitism rate, run type and fitness evaluation type. This information can be used for reference purpose and to determine the performance of the system.

The User can select a single-step run or a continuous run by using the run type option. In single-step run the user has to provide a start signal such as a button push after each iteration to proceed to next iteration. This allows the user to check termination criteria and other debug information as the system runs. This provides a kind of command on the system running process to the user. In a continuous run the system doesn't need anything from user and will run all the iterations without user interaction.

3.2.1.4 *Output Control Group*

Output Control Group parameters are default output. Parameters such as acceptable fitness, termination criteria also contribute to control output even though they are not listed among output control group. The default output of the system which is specified by the user is used as output if the fitness if the best chromosome is below the acceptable fitness value. For the autonomous car example, if none of the chromosomes have fitness above acceptable value it means that all of those chromosome actions may lead to damage to the car. In this case the default output is to stop the car so that no damage is done. If the fitness value of a chromosome is greater than the acceptable fitness then the output is set to that chromosome, not the default output. If there are multiple chromosomes with fitness above acceptable value then the output will be set to the best chromosome among them i.e. the chromosome with highest fitness value. If there

is more than one chromosome with same fitness above acceptable value then the most recently found chromosome will be selected.

3.2.1.5 *Fitness Function Group*

The fitness function group parameters are minimum fitness value, maximum fitness value and fitness evaluation type. There are two fitness evaluation types: internal and external evaluation. In an internal evaluation, the fitness function block (Fig 3.14) assigns a fitness value between the user specified minimum and maximum fitness value. Because the system is designed to accommodate a variety of systems, GPEAT allows different applications to be run using different maximum and minimum fitness values, not just 0 to 100. For example, some applications may have fitness boundaries between 0-10 where other applications may have fitness boundaries between 100-500. The GUI provides a window to enter minimum and maximum fitness values to the user. Acceptable fitness values must be between the minimum and maximum fitness values specified by the user.

3.2.1.6 *Sensor Group*

The sensor group parameters are the sensor number, sensor type, supply voltage and sensor output range. The number of sensors is specified and will change depending on the application. The sensor type can be set to analog or digital. This group includes the supply voltage and output ranges of each sensors. There are wide varieties of sensors. Examples include thermal sensors, velocity sensors, heat sensors and acoustic sensors etc. For example, autonomous car might use velocity sensors and a robot in space may use thermal sensors. Because different applications needs different sensors and each kind has

their own specifications (like voltage values, output ranges etc). These values should be provided by the user through the GUI.

All the parameters are summarized and listed in Table 3.1 on the basis of their requirements and dependences.

Table 3.1. GPeat parameters summary

Parameter	Group	Required	Dependent
Population size	Initial population generation	Yes	Iteration number
Chromosome size	Initial population generation	Yes	Gene size
Gene size	Initial population generation	Yes	Chromosome size
Gene type	Initial population generation	Yes	Chromosome size
Seed	Initial population generation	Yes	None
Increment	Initial population generation	Yes	None
Multiplier	Initial population generation	Yes	None
Modulus	Initial population generation	Yes	None
Operation type	Crossover and Mutation Group	Yes	Crossover type, mutation type
Crossover type	Crossover and Mutation Group	No	None
Mutation type	Crossover and Mutation Group	No	None
Crossover probability	Crossover and Mutation Group	No	None
Mutation probability	Crossover and Mutation Group	No	None
Arithmetic type	Crossover and Mutation Group	No	None
Crossover starting point	Crossover and Mutation Group	No	Crossover ending point
Crossover ending point	Crossover and Mutation Group	No	Crossover starting point

Elitism rate	System Control group	Yes	None
Termination criteria	System Control group	Yes	Acceptable fitness
Iteration number	System Control group	Yes	Population
Acceptable fitness	System Control group	No	None
Debug	System Control group	Yes	None
Run type	System Control group	Yes	None
Default output	Output Control Group	Yes	None
Minimum fitness value	Fitness Function Group	Yes	Maximum fitness value
Maximum fitness value	Fitness Function Group	Yes	Minimum fitness value
Fitness evaluation type	Fitness Function Group	Yes	None

3.2.2 Flow between Computer and Hardware

EA parameters are entered by user through GUI. Those EA parameters are mapped into a VHDL include file where those parameters are given names different from the field names used in the GUI. This include file is used to generate the main VHDL file. When generating the main VHDL file, the VHDL code is created by using the include file names to call the EA parameter values from include file. The main VHDL file is then placed in the Xilinx® file structure. Xilinx tools are used to partition, place/route design, generate bit file and finally download the design to the board. Xilinx® Spartan 3E™ is used as the GPeat programmable device. This flow is shown in Fig 3.13.

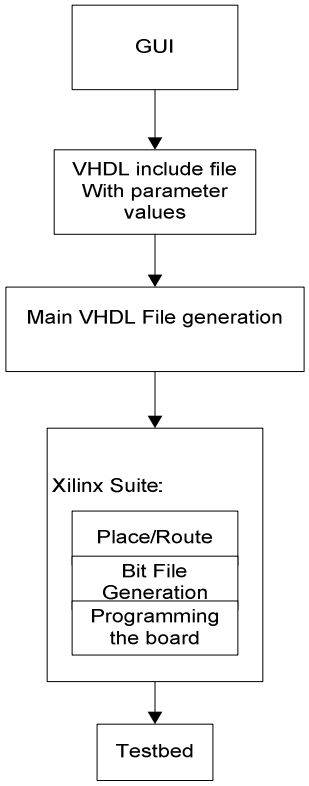

Figure 3.13 Software Flow

3.2.3 Hardware

The GPeat system is a testbed which implements the hardware part of the system on an FPGA. GPeat is a general purpose testbed which allows user to run most EA systems. This claim is supported by analysis of the patterns found in 40 to 50 [8-18] published papers on EA hardware systems and the information from books and experts. The pattern found through the analysis of these references allowed us to find a general purpose framework which all of the systems we examined could be reproduced. This EA flow can be generalized as shown in Fig 2.3.

Based on the EA parameters set through the GPeat user interface, the FPGA is programmed. The blocks shown in Fig. 3.14 can be divided into same six parameter

groups described in sections 3.2.2.1 through 3.2.2.6. The thicker arrows specify blocks whose function is determined by user input through the GUI. The function of the central control block is to control all the other blocks and is sequential logic. The other blocks are memory or largely combinational blocks and depend on the direction from the central control block. The functions of these blocks are explained below.

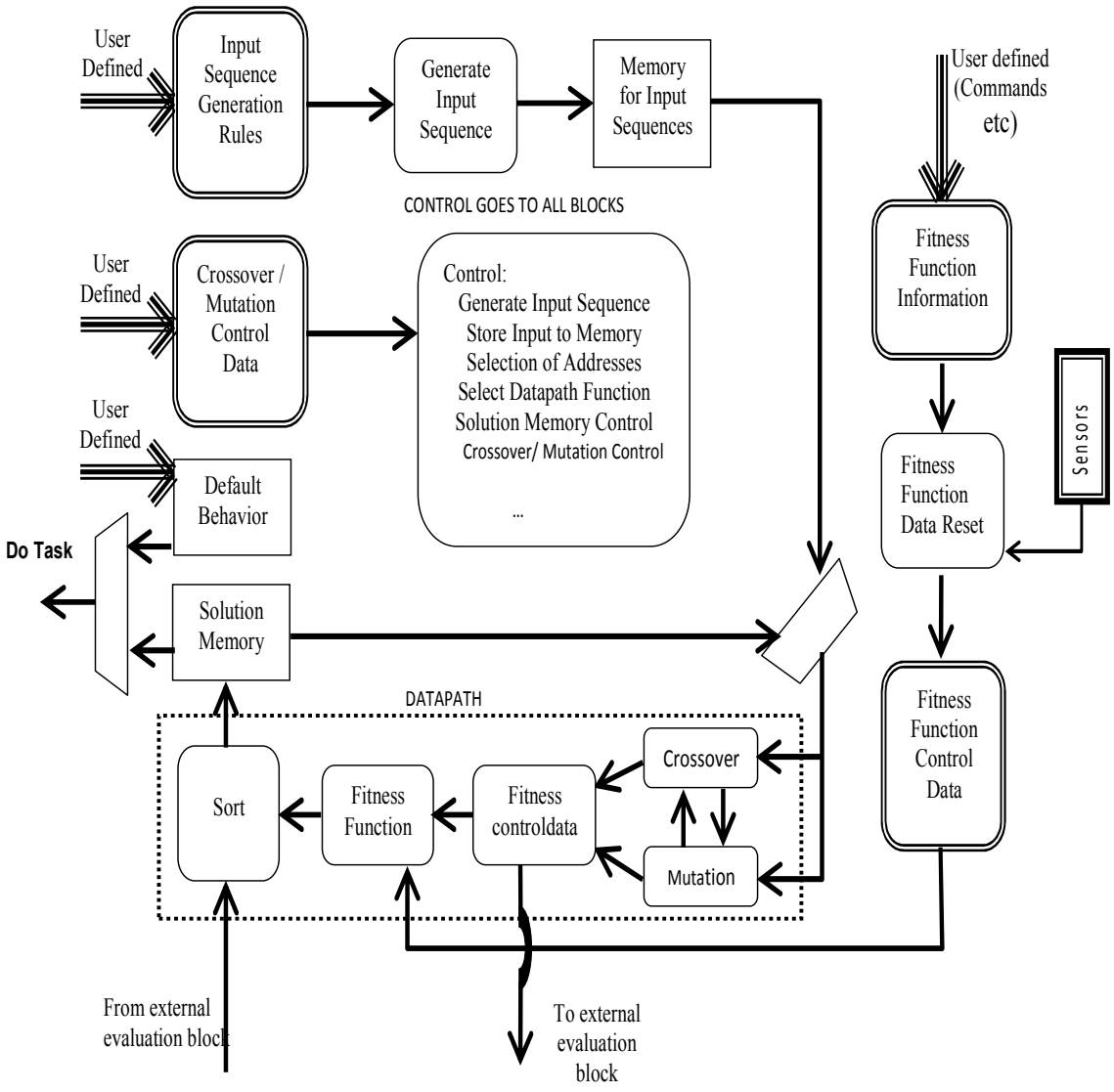

Figure 3.14 Hardware Block Diagram

3.2.3.1 *Initial Population Generation Blocks*

The initial population generation section of GPeat is made up of the input sequence generation rule database block, input sequence generation block and the memory for storing the generated chromosomes (Fig. 3.14). In the input chromosome rules generation block, rules provided by the user through the GUI are framed to generate initial chromosomes. The input sequence generation block will generate chromosomes based on input sequence generation block information. Here, one option is to use a random number generator to generate those chromosomes. The user may also specify a single value for all chromosomes or exact values for each chromosome. As an example of a case where all initial population individuals are all set to be equal (of population size 3) could be 1111, 1111, 1111 but, after a random flip bit mutation, they might become 1011, 1110 and 0111.These generated or user specified chromosomes form the initial population. The initial population size is given by the input parameter "population size". As the initial population is sent for reproduction a new set of chromosomes is produced for the subsequent generation. The user has an option of setting rules to check the eligibility of the chromosomes. In that case, the initial population will be a set of legal chromosomes. The system will show an error message if it does not find a legal chromosomes after specified number of iterations. Memory for the input sequence block stores initial population for that run, as all the chromosomes of initial population are not sent next block at once. The size of the memory is determined by the size of the population.

3.2.3.2 *Crossover and Mutation Control Blocks*

The crossover/mutation control information database block, crossover/mutation control block and crossover and mutation blocks (Fig. 3.14) are the blocks in this part of GPeat. The Crossover/mutation control data block has all the information from the GUI regarding the operations to be carried out upon generation of new chromosomes. It also determines the crossover type or mutation type to perform if user opts for exclusive reproduction type. The crossover /mutation control block is a small control block that controls the flow and flow of crossover and mutation blocks. In the crossover and mutation blocks, user specified crossover and mutations are carried out. GPeat provides a variety of crossovers and mutations as described in sections 3.2.1.2.1 and 3.2.1.2.2.

3.2.3.3 *Central Control Blocks*

The central control database block and central control block comes under this group. This group controls the flow of the GPeat. The central control data block holds the data entered by the user through GUI and is used to describe how the control block should run the system. The Central control block is connected to each and every block of the system shown in Fig. 3.14. Control signals are sent to blocks in order to activate or disable each block based on the information from the central control data block. First a control signal is sent to initial population generation blocks, after have initial population crossover and mutation blocks are signaled. Next fitness evaluation blocks are signaled (can be internal fitness evaluation block or external evaluation block based on user choose), followed by sorting block and finally the output blocks. For example, a control

signal will activate the internal evaluation block if user opts for it, otherwise the same control signal will disable that block if user opts for external evaluation.

3.2.3.4 *Output Control Blocks*

The sorting block, solution memory block and default output block comes under this group. This group's work is to sort all the solutions based on their fitness value and decide whether the default output or the best of the sorted solutions should be outputted. Chromosomes are also stored here for use in future generations. The sorting block carries out the sorting and the solution memory block stores the sorted chromosomes. The default output block holds the user specified default output and will output that upon receiving a control block signal which signifies no solution above the acceptable solution fitness value.

Sorting is the slowest procedure in many systems. The best of the sorting techniques works at O(n log n) [27]. In GPeat also, the sorting block was determined to be the slowest block of all the blocks to be designed. For example, the random number generator, crossover and mutation works at O(n). According to "Amdahl's law" to speed up the GPeat system, the sorting block must be improved. For that, we have replaced the sort hardware with three bins labeled 80%-100% solutions (bin1), 60%-80% solutions (bin2) and 0%-60% solutions (bin3). Solutions are stored in respective bins depending on their fitness percentage. For example, solutions with fitness percentage between 90-100 goes to 90%-100% bin in no special order. As our system gains a little by following traditional sorting techniques where each solution is ranked on other, we are following this type of sorting.

3.2.3.5 *Fitness Function Blocks*

All blocks related to fitness determination come under this group. The blocks are the fitness function information block, fitness function data reset block, fitness function control data block and fitness function calculation block. The main function of this group is to find a fitness value of chromosome based on user specified parameters. As explained in chapter1, fitness calculations can be done internally or externally. In an internal evaluation, the user will define rules which will be stored in the fitness function information block. Based on those rules and possibly information from the sensors, an objective function will be generated by the fitness function data reset block and fitness function control data block. The fitness function block will do the calculations of the fitness value according to the objective function. In an external evaluation, the candidate chromosome along with a start signal will be output to the external system where the calculation part may be carried out. When the fitness value is ready, the external block will send the GPeat system a done signal and it is GPeat's job to grab the fitness value at that time.

3.2.3.6 *Sensor Blocks*

The sensor blocks come under this group. The use of the sensor block is not available in this version of GPeat. The GPeat system breaks the barrier between the EA hardware systems and real-life events by providing a bread board area to connect any kind of sensors. This block will give the information from the environment supplied by the sensors to the fitness function data reset block. The sensor block collects the raw data from the external environment and then converts that data into format understandable by

system. This conversion is needed because of the wide variety of sensors. For example, if thermal sensors are used for a particular application, the output from those sensors will be analog and must be converted to binary through an on-board analog to digital converter. Once converted to a binary form the data can be used for processing.

3.2.4 Summary

As shown, the values entered through the GUI allow the user to set the behavior of GPeat without having to understand the hardware. Each block is configured according to the information entered by the user and the behavior may be change in moments to create a new system, making GPeat into an entirely different application. The next chapter will give details on the hardware design.

CHAPTER 4

VHDL DESCRIPTION

In this chapter a description of the VHDL code for implementing a general purpose evolutionary algorithm testbed is presented.

4.1 Top Level Design

This design is a Finite State Machine (FSM) which generates the individuals that make up a population, carries out reproduction, determines fitness of each individual and selects appropriate members for output and continuing into the next generation according to the user specified EA parameters. The state machine performs all the steps of a general purpose evolutionary algorithm described in Chapter 3.

Our FSM has nine states and the flow between those states is defined by the EA parameters entered by user. An Algorithmic State Machine (ASM) chart is used to show the detail implementation of the design. For explanation purpose ASM chart is divided into six parts (ASM-I to ASM-VI) in this chapter.

4.1.1 ASM Chart-I

This ASM chart performs the first step of our general purpose evolutionary algorithm which is the initial population generation. The initial population may be generated randomly using a random number generator or formed using some rules or

values supplied by the user. The random number generator we are using was explained in section 3.2.2.1.

This ASM chart consists of two states (state0, state1). State0 is a wait state; it will proceed to next state (state1) as soon as the program is downloaded on to the board. But in case of single cycle mode, after first iteration the transition from state0 to state1 depends on the input signal “start” which is triggered manually by a button push to a button on the Xilinx board. It will remain in state0 until “start” goes low irrespective of the number of clock cycles. It will proceed to next state (state1) on the immediate clock cycle, if “start” is high. State1’s function is to output a chromosome. For first iteration, that chromosome is generated according to user definition (either generated randomly or formed using some rules or values supplied). Later on, the chromosome from state1 depends on variable “percentage” (where “percentage” is the integer value of elitism rate). “Percentage” numbers of chromosomes are generated according to user definition and the rest are taken from previous chromosomes after first iteration. The last step of this chart is a dummy chromosome generation. Some reproduction types needs two chromosomes for processing. For that purpose a second “dummy” chromosome with user specified features is randomly generated. The ASM chart-I is shown in Fig 4.1.

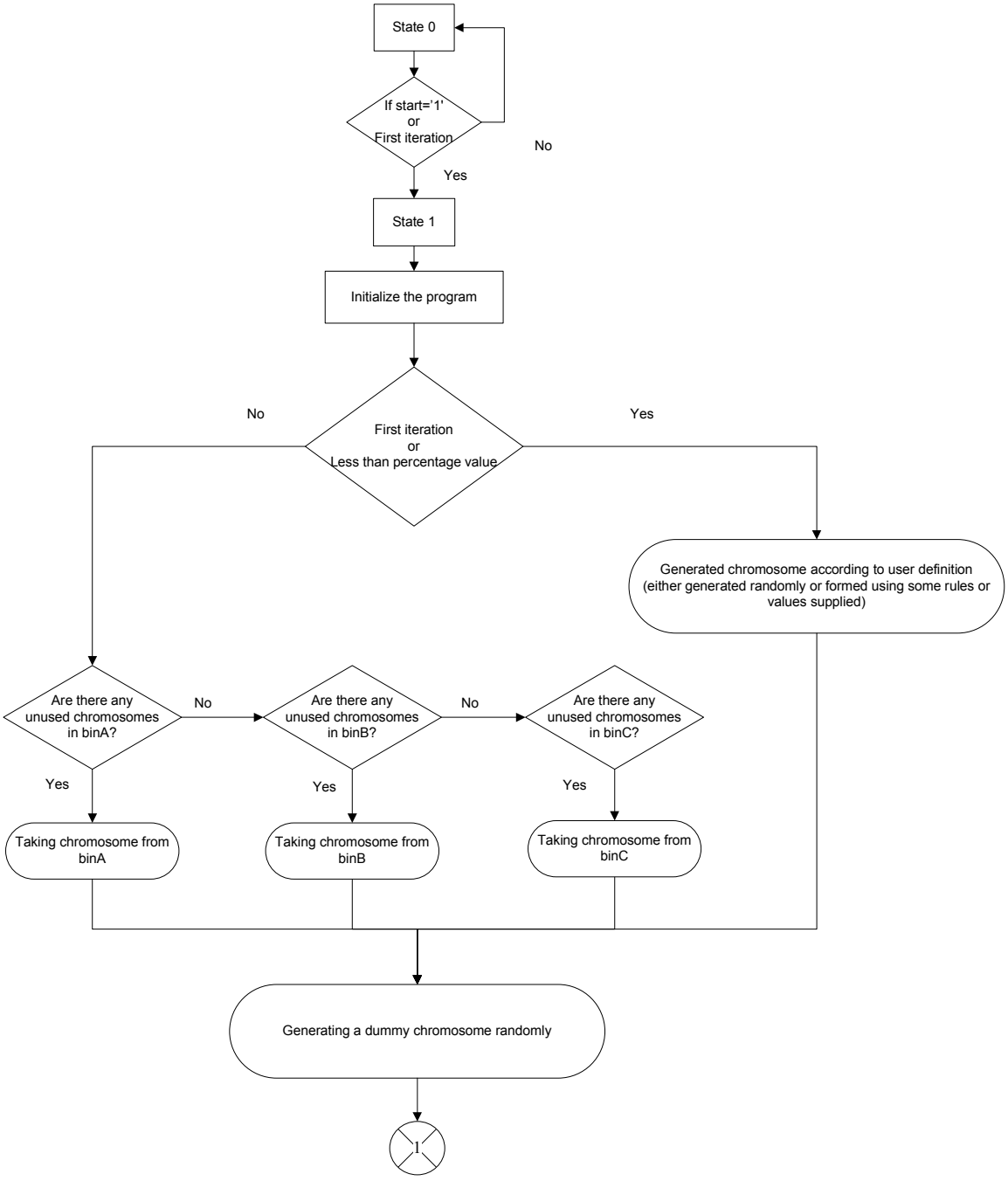

Figure 4.1 ASM chart-I.

4.1.2 ASM Chart-II

The ASM chart-II continues from the output of the ASM chart-I. ASM chart-II explains the first half of the reproduction process which consists of all crossover processing except type four (which is shown in the next ASM chart). The ASM chart-II is the second step of general purpose evolutionary algorithm.

This chart has only one state (state 2). A clock cycle is required for transition from state1 to state2. The type of reproduction to be performed is dictated by input value "C_M" can be "default", "crossover", "mutation", "both", "none" or "exclusive". If the reproduction type includes a crossover then one of the crossover types (explained in section 3.2.1.2) is executed on the chromosomes based on input value "T_C". The other required information for a particular crossover is provided by input values "K" and "P" where K and P are crossover starting point and crossover ending point respectively. There are 3 connectors in this chart which lead to ASM chart-III. The ASM chart-II is shown in Fig 4.2. All the reproduction types, crossover types and mutation types for different values of their parameters "C_M", "T_C" and "T_M" respectively are shown in Table 4.1.

Table 4.1 Reproduction, crossover and mutation types.

Integer value	C_M (Range 1-6)	T_C (Range 1-5)	T_M (Range 1-5)
1	Default crossover and mutation	Single point crossover	Flip bit mutation
2	Only crossover	Two point crossover	Exchange adjacent genes mutation

3	Only mutation	Uniform crossover	Mirror mutation
4	Both crossover and mutation	Arithmetic crossover	Flip a randomly selected bit mutation
5	None	Random shift genes crossover	Reciprocal exchange mutation
6	Exclusive		

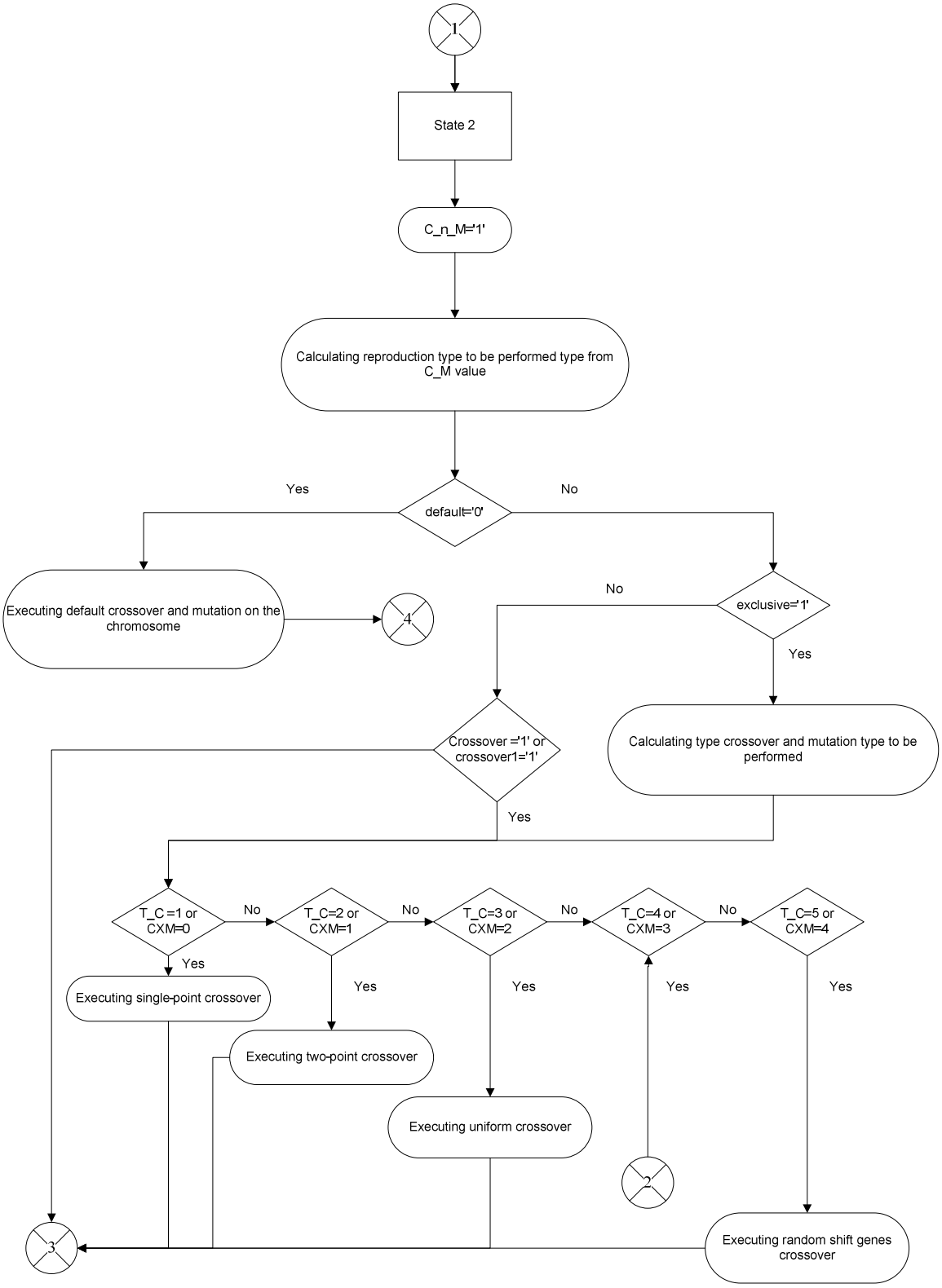

Figure 4.2 ASM chart-II.

4.1.3 ASM Chart-III

ASM chart-III along with ASM chart-II represents the second step of general purpose evolutionary algorithm. This chart describes the fourth type of crossover and the processing that is carried out during mutation.

There are no states in this chart as this chart along with ASM chart-II forms second state. The ASM chart-II and ASM chart-III processing are executed in single clock cycle. The sub-type of fourth crossover is executed based on input value "S_P" (where S_P specifies the sub type of arithmetic crossover). The output from crossover part is loaded to a signal named tmp_A if the reproduction type includes a crossover. Otherwise a chromosome from random number generator is loaded to tmp_A. If the reproduction type includes mutation then one of the mutation types (explained in section 3.2.1.2) is executed based on the value of "T_M". The output of the mutation block is loaded to signal "tmp_C" if mutation is a part of reproduction. Otherwise the chromosome generated in the crossover block is directly loaded to "tmp_C". If no reproduction takes place then a chromosome is directly loaded to "tmp_C" from the random number generator. The ASM chart is shown in Fig 4.3.

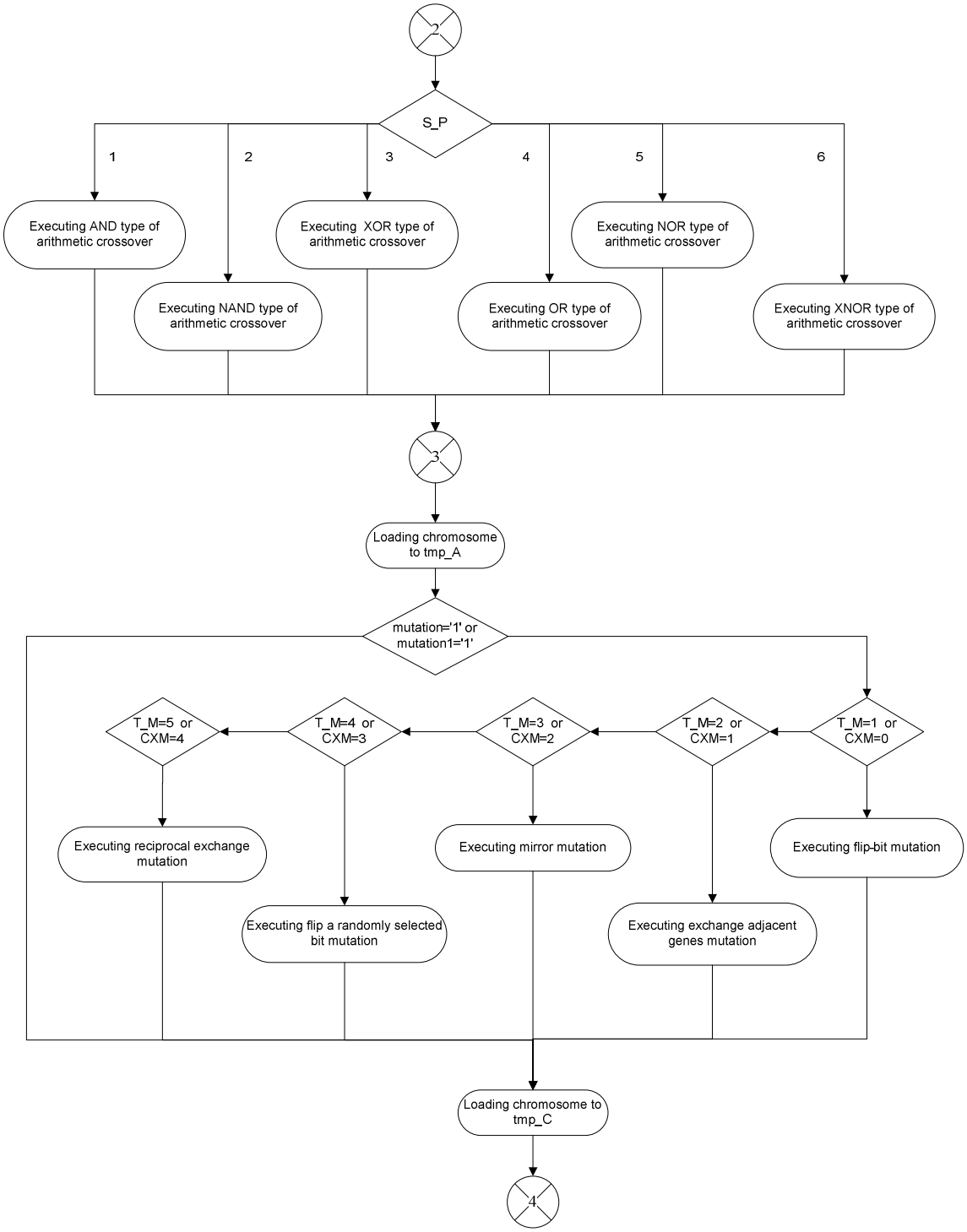

Figure 4.3 ASM chart-III.

4.1.4 ASM Chart-IV

ASM chart-IV continues from the output of the ASM chart-III. This ASM chart performs the next two steps of the general purpose evolutionary algorithm which are fitness calculation and storing. The fitness evaluation types are explained in detail in chapter 3.2.2.4.

Three states are involved in this chart and they are state3, state4 and state5 which are external fitness evaluation state, internal fitness evaluation state and storing state, respectively. If the input signal "IrE" is high it means that fitness evaluation is to be done externally and the internal fitness value rules are bypassed. This signal is set through the GUI by specifying an intrinsic or extrinsic run. When a chromosome is ready to be evaluated a start signal "start1" is outputted to an external fitness calculating system along with chromosome named "chr" and the GPeat code will move to next state (state3) from state2. State3 is a wait state that will wait until a done signal, "done_E", is received from the external fitness calculating system. The external evaluation system must also return the fitness value "fit" on the same clock edge as it returns the "DONE" signal. GPeat can wait in this state an infinite number of clock cycles since a signal from the external block is required for GPeat to continue on to the next task. Once "done_E" is high it will move to next state (state5) on the next clock cycle. If "IrE" is low (means internal evaluation), it will move to state4 from state2 on a single clock cycle. Here fitness is taken from a pre-calculated array using the chromosome itself as an index and from there it will move to state5 on next clock cycle. The pre-calculated values stored for use in internal evaluations can be generated from equations or rules. For example, if a fitness is defined as FITNESS = $Gene1^2$, then the fitness value is calculated and stored at

an address equal to gene 1. To find the fitness value using this method, gene1 is broken out from the chromosome under evaluation and used as an address to access the pre-calculated fitness. State5 is storing state, where each chromosome will be stored in their respective bin based on their fitness values as explained in section 3.2.3.4. Here user inputs “max” and “min” are used calculate the boundaries of those bins which are bin1, bin2 and bin3 respectively, explained in detail in section 3.2.3.4. The ASM chart-IV is shown in Fig 4.4.

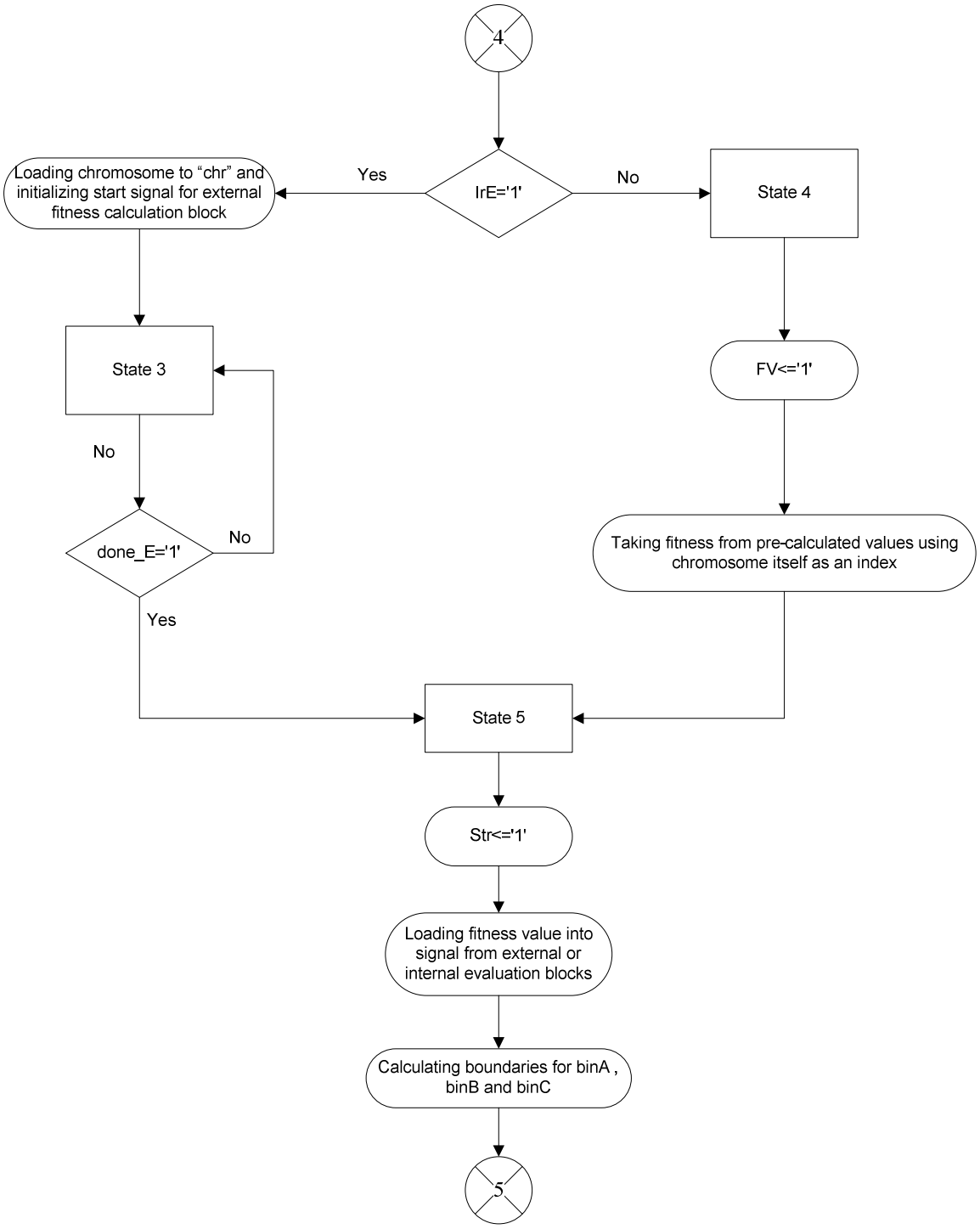

Figure 4.4 ASM chart-IV.

4.1.5 ASM Chart-V

ASM chart-V continues from the output of ASM chart-IV. This chart completes the storing process and performs debugging if user opts for it. The debugger is explained in detail in section 3.2.2.4.

In this ASM chart the remaining part of state5 will be implemented and then it will move to the next state which can be state6, state1 or state8. If the input signal "debugger" is high the next state is state6. State6 is a wait state; it will remain in this state if the input signal from debugger, "busy", goes high. Once the "busy" signal goes low, GPeat will move to next state (state7) on the next clock cycle where it will output all the necessary data needed for debugging process to the computer. Up on next clock cycle it will go to state8 or state1. It will go to state8, after completing every iteration otherwise it will go to state1. In state8, it will update all variables and signals that are used for internal processing of VHDL code. If the "debugger" signal is low then it will directly move to state8 or state1 checking point. The ASM chart-V is shown in Fig 4.5.

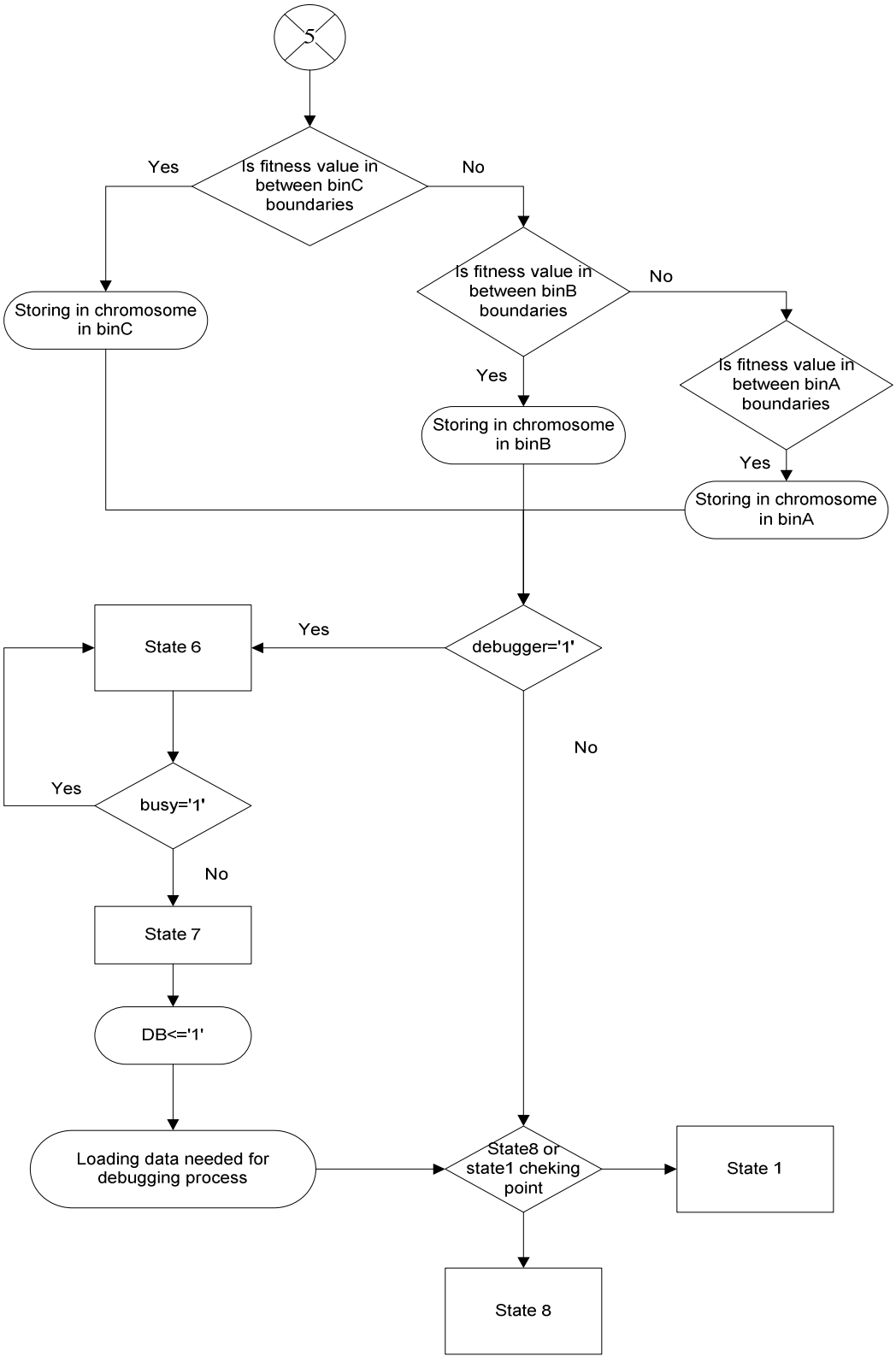

Figure 4.5 ASM chart-V.

4.1.6 ASM Chart-VI

ASM chart-VI follows from the output of ASM chart-V. This chart implements the done state where the termination criteria comes into play. The termination criteria types are explained in detail in section 3.2.1.3.

This chart starts with state8 , the final state, and explains it's flow to other states based on input signal "S_C" ("S_C" specifies the type of termination criteria explained in section 3.2.1.3). First all the variables (used for internal processing of VHDL code) are reset to their initial values and then based on the used defined acceptable fitness value "AF" the output signal "data_out" is loaded. The fitness of best chromosome up to that point is compared with "AF". If the fitness of that chromosome is greater than "AF" then the best chromosome will be outputted through the output signal "data_out". Otherwise the output will be "default_out" which is the value that the user gave to the system through the GUI to say what the default output should be if an appropriate output has not been found yet. Up on the next clock cycle it will go to state0, if it is in single cycle mode or it completes all iterations in continuous cycle mode otherwise it will go to state1. The ASM chart-VI is shown in Fig 4.6.

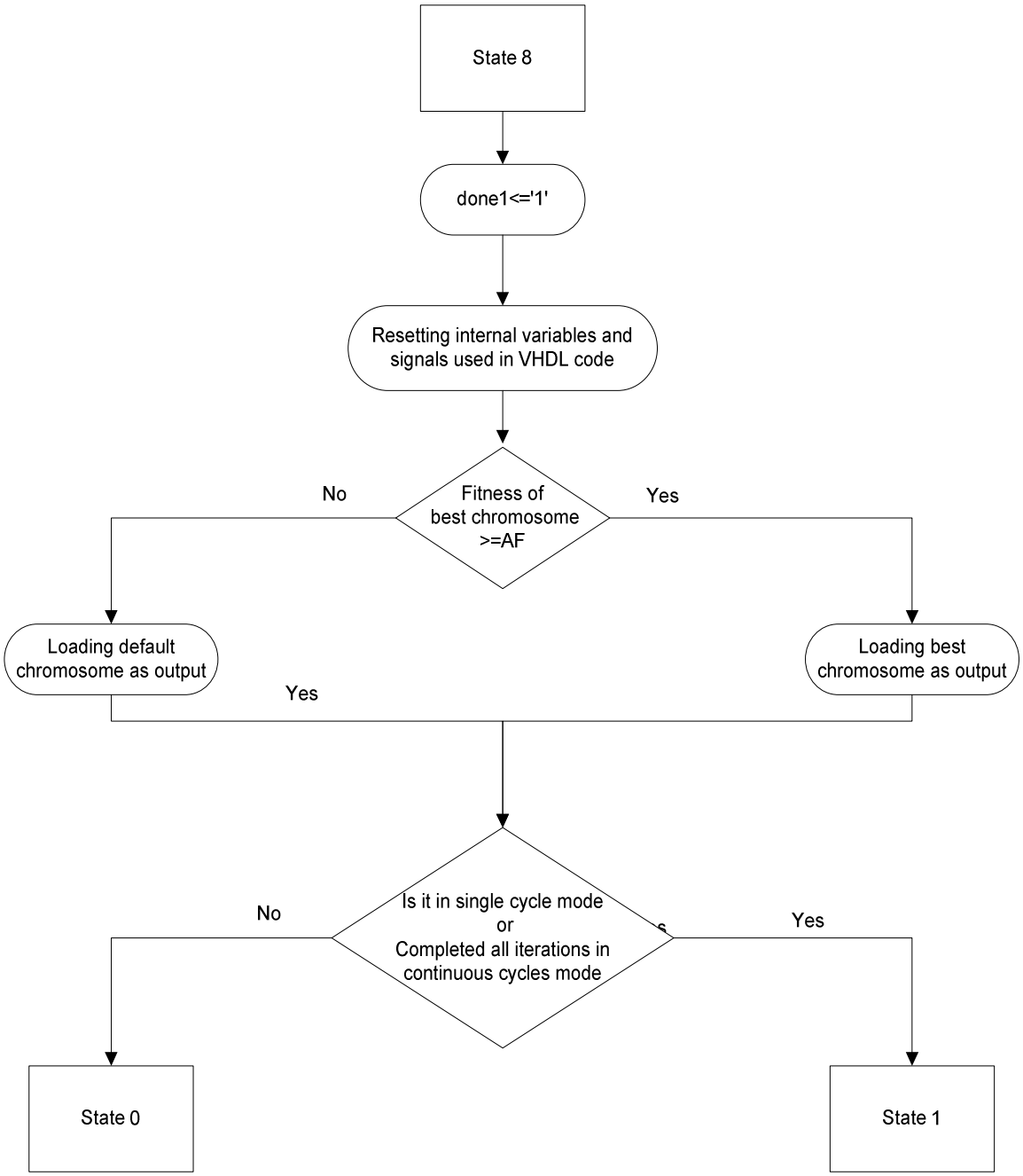

Figure 4.6 ASM chart-VI.

If input signal “Singlestep” is high (which means that a single cycle run has been selected by the user), after each iteration it will wait for a “start” signal form user in state0 to move to next iteration until it meets user specified kind of termination criteria. A start signal is, for example a button push on the board. Once the termination criteria, stops after coming to state0 irrespective of “start” signal. If “singlestep” is low (means continuous run), it comes to state0 only after meeting the termination criteria and stops there.

These six ASM charts (ASM-I to ASM-VI) explain the implementation of our general purpose evolutionary algorithms design. As the flow between the states depends on user specified EA parameters, the total number of clock cycles required will not be fixed.

CHAPTER 5

EXPERIMENTAL RESULTS

In this chapter we implement a color search algorithm on GPeat and analyze and compare the simulated and synthesized (hardware implementation) results. The color search application is run as an intrinsic application where its fitness value determination is done by a block external to GPeat but on the same Xilinx chip as GPeat is run

5.1 Color Theory

Color theory describes the mixing of colors and the visual impacts of the different color combinations. The "Additive" and "Subtractive" models are the two basic color models. The additive model produces colors by mixing the three primary colors, red, blue and green. The subtractive model produces colors by mixing paints, pigments, inks and dyes with red, yellow and blue as the basic colors [29].

We use the subtractive model of color theory in a color search application to test the Gpeat system but the example can easily be extended to include the additive model. The secondary colors orange, green and violet are created from primary colors (red, yellow and blue). Different amounts of secondary colors are mixed to produce, for example, the tertiary colors red-orange, yellow-orange, yellow-green, blue-green, blue violet and red-violet. In this application, we have four target colors gold, red-orange,

copper and blue-green. The main concept of a color search is the evolution to secondary or tertiary colors from basic colors. This color search shows a GPeat search of a 3D solution space and the debugger abilities.

5.2 Description of Experiments

For this application we conducted four different experiments, each showing how, generation by generation, GPeat will get closer to a finding a color near one the target colors. Each experiment uses different extrinsic block (real test environment, external block). These extrinsic blocks differ with each other in the objective function used for finding fitness which is based on target color. Each basic color is represented as a gene and the level of each color is expressed using 3 bits which is the gene size. The length of each chromosome for this application is therefore 3 bits x3 genes i.e., 9 bits. As the gene size is 3 bits, each gene can have a maximum of level 7 of that particular color and a minimum level of 0. This means that the maximum level of color can be 7+7+7=21 and the minimum level of color can be 0+0+0=0. These values are considered as maximum and minimum values of fitness for this application. These values will not change even if the color being searched for does and therefore is the same for all experiments of this application.

Except the above mentioned values, the remaining system parameters are not fixed for the system and can be same or different for each experiment. In the experiments described in this section, different sets of input values are used so as to test a variety of scenarios. GUI provided by GPeat allows user to set all the parametric values needed for these experiments. To make the results easier to understand, a relatively small population size and iteration number has been used. Many of the documented systems have a

population in the hundreds and also have hundreds of generations. The set of input values for each experiment are shown in table 5.1.

5.2.1 Experiment 1

In first experiment, our target color is gold which requires 3 parts of red, 6 parts of yellow and 1 part of blue [30]. This experiment uses an extrinsic block with an objective function shown in equation 5.1 for finding fitness.

$$fitness = maximum\ fitness - (|3 - parts\ of\ red| + |6 - parts\ of\ yellow| + |1 - parts\ of\ blue|) \quad (5.1)$$

The absolute value of the difference between the actual value and the ideal value (here ideal values are 3, 6 and 1 for red, yellow and blue respectively) tells how far the actual value is from the ideal value on a scale of 0 to 7 (i.e. how far that gene is from the ideal gene). The sum of the absolute values tells how far the color mixture is from the ideal color on a scale of 0 to 21 (i.e. how far the chromosome is; from ideal chromosome). By subtracting this sum from maximum fitness value gives you how close the color mixture is from the ideal color (gold). The set of input values used for this experiment are shown in Table 5.1.

5.2.2 Experiment 2

In our second experiment the target color is red-orange which requires 3 parts of red, 2 parts of yellow and no blue [30]. This experiment uses an extrinsic block with an objective function shown in equation 5.2 for finding fitness.

$$fitness = maximum\ fitness - (|3 - parts\ of\ red| + |2 - parts\ of\ yellow| + |0 - parts\ of\ blue|) \quad (5.2)$$

The ideal gene values in this case are 3, 2 and 0 for red, yellow and blue respectively.

A different set of input values are used in this experiment as shown in Table 5.1.

5.2.3 Experiment 3

In this experiment the target color is copper which requires 1 parts of red, 2 parts of yellow and 1 part of blue [30]. This experiment uses an extrinsic block with an objective function shown in equation 5.3 for finding fitness.

$$fitness = maximum\ fitness - (|1 - parts\ of\ red| + |2 - parts\ of\ yellow| + |1 - parts\ of\ blue|) \quad (5.3)$$

The ideal gene values in this case are 1, 2 and 1 for red, yellow and blue respectively.

A different set of input values are used in this experiment from the first two experiments as shown in Table 5.1.

5.2.4 Experiment 4

In this experiment the target color is blue-green which requires 0 parts of red, 1 parts of yellow and 2 part of blue [30]. This experiment uses an extrinsic block with an objective function shown in equation 5.4 for finding fitness.

$$fitness = maximum\ fitness - (|0 - parts\ of\ red| + |1 - parts\ of\ yellow| + |2 - parts\ of\ blue|) \quad (5.4)$$

The ideal gene values in this case are 0, 1 and 2 for red, yellow and blue respectively.

A different set of input values are used in this experiment from previous experiment as shown in Table 5.1. The input values for all four experiments are listed in Table 5.1.

Table 5.1 Input parameters set for all experiments of this application

Parameter	Experiment1	Experiment 2	Experiment 3	Experiment 4
Gene type	Binary	Binary	Binary	Binary
Population	10	20	12	7
Iterations	10	4	7	7
Elitism rate	50%	40%	84%	72%
Operation type	Default	Both	None	Exclusive
Type of cross over	No need to specify	Arithmetic	No need to specify	Arithmetic
Type of mutation	No need to specify	Mirror	No need to specify	Mirror
Arithmetic type	No need to specify	AND	No need to specify	Nor
Cross over starting point	No need to specify	No need to specify	No need to specify	0
Cross over ending point	No need to specify	No need to specify	No need to specify	1
Crossover random gene number	No need to specify	No need to specify	No need to specify	2
Mutation random gene number	No need to specify	No need to specify	No need to specify	2
Acceptable fitness	17	18	15	20

Seed	6	7	3	4
Multiplier	5	5	5	5
Increment	3	7	5	1
Modulus	8	8	8	8
Termination criteria	Iteration	Fitness	Fitness	Iteration
Debug	Yes	No	No	Yes
Run type	Continuous	Single cycle	Single cycle	Continuous
Fitness evaluation type	External	External	External	External
Default output	(000000000)	(111111111)	(000000000)	(111111111)

5.3 Simulation Setup

To verify the design, the experiments described in section 5.2 are selected for simulation. The hardware results will follow in the next section. The best chromosome for given parameters is obtained from hardware simulation using a GPeat system with values set through the GUI and run on Xilinx ISE using the ModelSim XE-III (MXE-III) simulator.

During simulations the inputs, clock and start are given manually through simulator. The simulation starts on a clock signal rising edge when the “start” signal is high (start signal automatically stays high when the board is programmed for a clock cycle in hardware execution). “start” will go low after one clock cycle. Each state takes

one clock cycle for execution except the wait states. The total number of clock cycles required also depends on parameters run type, whether debugging is turned on and the evaluation type. So the number of clock cycles required for a complete run until termination criteria has been met can be calculated using the formulas given below.

$$N = 1 + (1 \times S) + \big(M \times I \times (3 + E + D)\big) + I + X \tag{5.5}$$

Where:

M = population size

I = number of iterations executed.

$$S = \begin{cases} 1 \text{ for continous} \\ I \text{ for single step} \end{cases}$$

$$E = \begin{cases} 1 \text{ for internal evaluation} \\ \text{number of external evaluation block clock cycles} \end{cases}$$

$$D = \begin{cases} 0 \text{ without debugger.} \\ 1+ \text{ number of debugger wait state clock cycles} \end{cases}$$

$$X = \begin{cases} 0 \text{ for continous run type} \\ (\text{iterations-1}) + \text{sum of clock cycles in state0 for single step run type} \end{cases}$$

For simulations the number of cycles taken by the debugger and the external evaluation blocks must be manually set and was set to one for these simulations. The number of clock cycles required can be calculated from above formula.

For this application, the chromosome size is 9 bits . The size of the bins (arrays) used in this system for sorting are set during VHDL code generation and are of the size

$$\text{Population} \times \text{Iterations}$$

i.e., for experiment1, arrays are 100×9 bits. For experiment 2, arrays are 80×9 bits. For experiment 3 arrays are 84×9 bits and for experiment 4 they are 49×9 bits.

5.4 Simulation Results

The data from the simulation results for all four experiments is shown in Table 5.2. We used a clock with a period of 0.1ns for simulations. In these simulations, the debugger wait state takes only one clock cycle and after the first iteration GPeat stops and waits for a clock cycle in start state. Those values are used while finding the number of clock cycles from Eqn. 5.5.

Table 5.2 Simulation results

Data	Experiment 1	Experiment 2	Experiment 3	Experiment 4
Target Chromosome	(011110001)	(011010000)	(001010001)	(000001010)
Output chromosome	(010110010)	(011001000)	(010001001)	(111111111)
Target fitness	17	18	15	20
Fitness achieved	17	19	16	--
Executed iterations	10 out of 10	2 out of 4	1 out of 7	7 out of 7
Total time	61.2 ns	16.6 ns	5.1 ns	30.3 ns

Number of chromosomes in 80-100% bin	100-73 = 27	160-146 = 14	84-81 = 3	49-41 = 8
Number of chromosomes in 60-80% bin	100-53 = 47	160-141 = 19	84-75 = 9	49-16 = 33
Number of chromosomes in 0-60% bin	100-74 = 26	160-153 = 7	84-84 = 0	49-41 = 8
Executed chromosomes	100	40	12	49

The chromosomes in each bin after simulation for experiments (1-4) are shown in Table 5.3. The number of chromosomes in a bin is calculated by finding the number of spaces left in the bin through simulation and then by deleting that number from actual array length and sum of those numbers of bins should be equal to executed chromosomes number.

Table 5.3 Chromosomes in bins for all experiments.

Experiment 1	Bin 1	(001010011)2 (010110010)2 (110100000)2 (110010000)3 (010010100)3 (010110010)5 (110010000)5 (110100000)6 (110010000)7 (110011000)8 (010110010)9 (110011000)9
	Bin 2	(111011001)1 (001101011)1 (111011001)1 (1001101011)1 (111011001)2 (001101011)2 (111011001)2 (010100100)2 (100100110)2 (000100010)3 (010110100)3 (100000110)3 (000100010)3 (111101001)3 (001100011)3 (100011110)3 (011011101)3 (111011001)4 (001101011)4 (000011010)4 (111001001)4 (001011011)4 (100010110)4 (000100010)5 (010110100)5 (100000110)5 (000011010)5 (111011001)5 (010100100)5 (001100011)5 (111011001)6 (001101011)6 (101010111)6 (000100010)6 (111110001)6 (010100100)6 (100000110)7 (000100010)7 (010110100)7 (100000110)7 (011101101)7 (110101000)7 (000001010)7 (001101011)8 (111011001)8 (001101011)8 (100100110)8 (011100101)8 (000100010)8 (010110100)9 (100000110)9 (000100010)9 (010110100)9 (001101011)9 (100010110)9 (011101101)9 (111011001)10 (001101011)10(111011001)10(010011100)10 (100011110)10 (011001101)10 (110111000)10

	Bin 3	(011111101)1 (101001111)1 (011111101)1 (101001111)1 (011111101)1 (101001111)1 (011111101)2 (101001111)2 (101001111)4 (011111101)4 (101001111)4 (010111100)4 (101101111)5 (011111101)6 (101001111)6 (011111101)6 (101011111)7 (111111001)7 (011111101)8 (101001111)8 (101011111)8 (101011111)9 (011111101)10 (101001111)10 (001111011)10
Experiment 2	Bin 1	(001100000)1 (001100000)1 (001100000)1 (001100000)1 (001100000)1 (001100000)2 (001100000)2 (000000000)2 (011000000)2 (000000000)2 (011010000)2 (000000000)2 (010000000)2 (010000000)2
	Bin 2	(000101000)1 (000101000)1 (001001100)1 (000101000)1 (001001100)1 (000101000)1 (001001100)1 (000101000)2 (001001100)2 (000101000)2 (001001100)2 (000000011)2 (000000001)2 (000000001)2 (000000001)2 (000000001)2
	Bin 3	(000000101)1 (000000101)1 (000000101)1 (000000101)1 (000000101)1 (000000101)2 (000000101)2
Experiment 3	Bin 1	(010001100)1 (010001100)1 (010001001)1
	Bin 2	(100011110)1 (110101000)1 (000111010)1 (100011110)1 (110101000)1 (000111010)1 (100011110)1 (110101000)1 (000111010)1
	Bin 3	---
Experiment 4	Bin 1	(000100100)2 (000011000)2 (000100100)2 (000100100)3 (000100100)5 (000100100)5 (000011000)6 (000011000)6
	Bin 2	(100101010)1 (100001000)1 (000101000)1 (000111100)1 (101010010)1 (100000000)1 (001100000)1 (100001000)2 (000101000)2 (000111100)2 (101010010)3 (100000000)3 (001100000)3 (101010011)3 (100001000)4 (000101000)4 (000111100)4 (101010010)4 (100000000)4 (101000101)4 (100000100)4 (001100100)5 (100101000)5 (001101000)5 (101010010)6 (100000000)6 (001100000)6 (100001000)7 (000101000)7 (000111100)7 (101010010)7 (101010011)7
	Bin 3	(001110110)2 (001110110)3 (001110111)3 (001110110)5 (110101000)5 (100111100)6 (001110110)7 (001110111)7

Tables 5.2 and table 5.3 shows the entire simulation data. In table 5.3 the output chromosome in bin1 of each experiment is masked with gray color and each chromosome

is placed in brackets and indexed (outside the bracket) with its iteration number for easy illustration. As the output chromosome of experiment 4 is default value we can't find a masked chromosome in bin1 of experiment 4. This data should match the synthesis data so as to check for simulation and synthesis mismatch. The screen shots of simulations for all experiments are presented below.

All these simulations are carried out on ModelSim XE-III (MXE-III) simulator. As screen shot of simulation process for all experiments look similar, the screen shot of experiment 1 is shown below as an example. We can observe all the signals on the left side and their corresponding values or waves on the right side of the screen shots. Clock and start on the top of signals list are the input signals and data_out is the output signal while rest is internal signals displayed in screen shot. We can also observe the clock pulse on the top and total time on the bottom of the screen shot. Fig 5.1 shows the simulation screen shot of experiment 1 whose target color is gold.

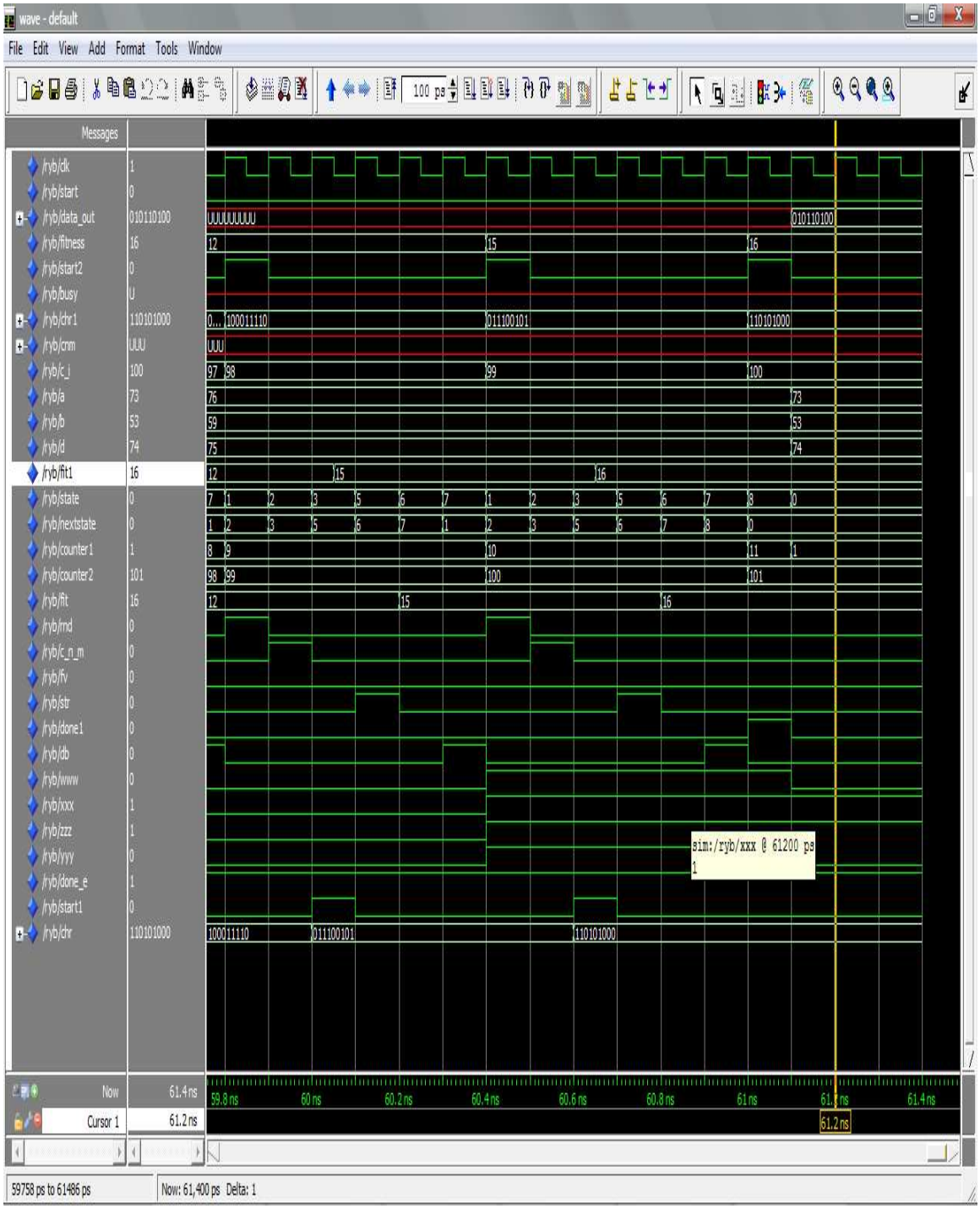

Figure 5.1 Simulation screen shot of experiment 1.

5.5 Synthesis, Timing, Design Implementation and Utilization

We used Xilinx ISE 9.2i design suite to program the board. Xilinx ISE 9.2i design suite features synthesis through Xilinx® Synthesis Technology (XST), static timing through Xilinx's timing analyzer, design implementation (mapping, place-and-route) and creation of bit files. We used Xilinx® Spartan 3E™ starter kit as our testbed programmable device for all of our experiments.

After synthesis, the inputs and outputs of the circuit are mapped to pins on the Xilinx chip by specifying the signal and pin ID in the user constraints *file* (UCF file). IO mapping is shown below in the form that is used in the UCF for GPeat. The pins on a Xilinx chip are specified by using a character, A through T, to specify a row and a number, 1 through 16, to specify a column as shown in Fig 5.2.

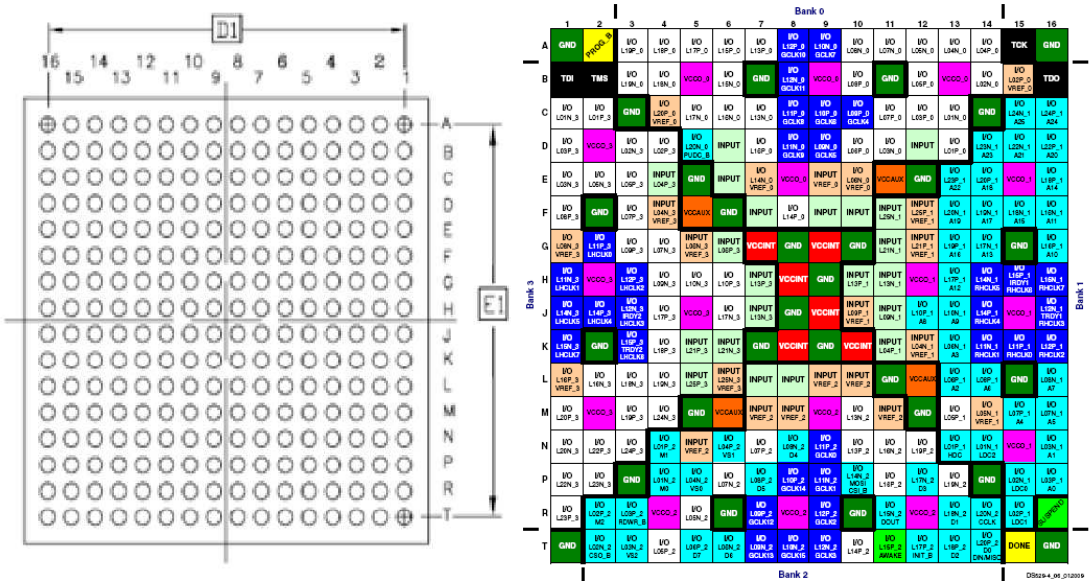

Figure 5.2 Ball grid array and pin package footprint.

The ball grid array is shown in figure 5.2. Ball grid array is a package where pins are soldered in form of a grid through which electric signals are sent from integrated circuits to printed board. The top view of package foot print is also shown in Fig 5.2.

User Constraints File (.UCF):

```
NET "clk" LOC = "C9";
NET "start" LOC = "D18";
NET "reset" LOC = "K17";
NET "tx_intf" LOC = "M14"  ;
NET "rx_intf"  LOC = "R7"  ;
```

We used 50 MHz clock provided on the board. “Start” and “reset” are connect to push buttons where “reset” is used by debugger to reset the debugging process. “tx_intf” and “rx_intf” are connected to DCE female DB9. From Fig 5.2 we can locate the pins assigned to the input signals on ball grid array.

Once the IO is specified the bit file can be generated. A bit file is the configuration file that is generated by the Xilinx tools and is downloaded to the Xilinx board to program the board to behave according to the circuit description given by the designer.

Debugger is used to take the values from board to computer. As a part of checking all the possible options of GPeat, debugger is used in experiment 1 and experiment 4 while experiment 2 and experiment 3 doesn’t. The debugger uses the RS-232 protocol (Recommended Standard 232) to pass data between the board and the computer in serial, binary data transfers. The chromosomes of each generation along with their fitness values are sent back to the computer and stored. The debugger can store chromosomes both in binary and integer forms. The debugger doesn’t put chromosomes in different bins according to their fitness; it will just store chromosomes at same place along with their

fitness values under other EA parameters as header. As we are using the debugger only for experiments 1 and 4, the chromosomes collected through debugging process along with their fitness values of only experiments 1 and 4 are shown in table 5.4. The debugger stores chromosomes iteration wise and to distinguish them every alternate iteration chromosomes and their corresponding fitness values are colored yellow and output chromosome with gray color in table 5.4.

Table 5.4 Chromosomes from debugger

	Chromosomes	Fitness values
Experiment 1	(011111101 101001111 111011001 001101011 011111101 101001111 111011001 001101011 011111101 101001111)1	16,8,14,16,16,8,14,16,16,8,
	(111011001 001101011 011111101 101001111 111011001 010100100 001010011 100100110 011010101 110100000)2	14,16,16,8,14,11,13,13,15,15,
	(000100010 010110100 100000110 110010000 000100010 111101001 010010100 001100011 100011110 011011101)3	15,17,9,13,15,11,15,16,12,14,
	(101001111 111011001 001101011 011111101 101001111 000011010 111001001 010111100 001011011 100010110)4	8,14,16,16,16,13,11,14,14,14,
	(110010000 000100010 010110100 100000110 110010000 101101111 000011010 111011001 010100100 001100011)5	13,15,17,9,13,9,14,14,15,15,
	(011111101 101001111 111011001 001101011 011111101 110100000 101010111 000100010 111110001 010100100)6	16,8,14,16,16,11,8,13,15,16,
	(100000110 110010000 000100010 010110100 100000110 011101101 110101000 101011111 000001010 111111001)7	9,13,15,17,13,13,15,11,13,16,
	(001101011 011111101 101001111 111011001 001101011 100100110 011100101 110011000 101011111 000100010)8	16,16,8,14,16,15,12,11,10,16,
	(010110010 100000110 110010000 000100010 010110100 001101011 100010110 011101101 110011000 101001111)9	17,9,13,15,16,13,9,15,15,10,
	(111011001 001101011 011111101 101001111 111011001 010011100 001111011 100011110 011001101 110111000)10	14,16,16,8,14,13,12,12,15,16

Experiment 4	(100101010 100001000 000101000 000111100 101010010 100000000 001100000)1 (000100100 001110110 100001000 000101000 000111100 000011000 000100100)2 (101010010 100000000 001100000 000100100 001110110 101010011 001110111)3 (100001000 000101000 000111100 101010010 100000000 101000101 100000100)4 (001100100 000100100 001110110 100001000 000101000 001101000 110101000)5 (100111100 101010010 100000000 001100000 000100100 000011000 000011000 001110110)6 (100001000 000101000 000111100 101010010 001110111 101010011)7	13,15,15,13,15, 14,15,16,11,15, 15,13,17,16,15, 14,15,16,11,14, 10,15,15,13,15, 14,12,14,15,16, 11,15,15,14,9, 9,15,14,15,16, 17,17,11,15,15, 13,15,10,14

5.6 Analysis

In this section, the total time taken and hardware usage for all four experiments are analyzed with the help of simulation and synthesis results. The resource utilization and distribution for the design is shown in Table 5.5 and timing constraints are shown in Table 5.6.

Table 5.5 Design Summary

		Actual number	Used			
			Exp 1	Exp 2	Exp 3	Exp 4
Logic utilization	Total Number Slice registers • used as Flip Flops • used as Latches	9,312	889 (9%) 856 33	574 (6%) 559 15	1264 (13%) 1,250 14	1,045 (11%) 1,012 33
	Number of 4 input LUTs	9,312	1,343 (14%)	916 (9%)	1,559 (16%)	1,427 (15%)

Logic Distribution	Number of occupied Slices • only related logic • unrelated logic	4,656	1,114 (23%) 1,114 0	731 (15%) 731 0	1,416 (30%) 1,416 0	1,197 (25%) 1,197 0
	Total Number of 4 input LUTs • used as logic • used as a route-thru • used for Dual Port RAMs	9,312	1,628 (17%) 1,343 253 32	1,158 (12%) 916 220 22	1,799 (19%) 1,559 218 22	1,711 (18%) 1,427 252 32
	Number of bonded IOBs • IOB Flip Flops	232	81 (34%) 48	81(34%) 11	81(34%) 11	81(34%) 52
	Number of GCLKs	24	4 (16%)	3(12%)	3(12%)	4 (16%)
	Number of MULT18X18SIOs	20	1 (5%)	1 (5%)	1(5%)	
	Total equivalent gate count for design		21,070	14,509	24,091	22,794
	Additional JTAG gate count for IOBs		3,888	3,888	3,888	3,888

Table 5.6 Timing summary

Timing Constraints with speed grade -5				
Minimum period (Maximum frequency)	13.259ns (75.422MHz)	12.784ns (78.220MHz)	11.948ns (83.6MHz)	13.786ns (72.538MHz)
Minimum input arrival time before clock	4.809ns	4.809ns	4.809ns	4.809ns
Maximum output required time after clock	4.754ns	4.040ns	4.040ns	4.754ns
Maximum combinational path delay	No path found	No path found	No path found	No path found

5.6.1 Timing

The total number of clock cycles required for simulations are 612, 166, 51 and 303 respectively for experiment1, experiment2, experiment3 and experiment4 see Table 5.2. By using equation 5.5 we can calculate the actual number of clock cycles required.

Table 5.7 Clock cycles analysis table

	Calculated clock cycles $N = 1 + (1 \times S) + \big(M \times I \times (3 + E + D)\big) + I + X$	Clock cycles from simulation results
Experiment 1	N = 1+ 1×1 + 10×10×3+1+2 + 10 = 612	612
Experiment 2	N = 1+ 1×2 + 20×2×3+1+0 + 2+1 = 166	166
Experiment 3	N = 1+ 1×1 + 12×1×3+1+0 + 1+0 = 51	51
Experiment 4	N = 1+ 1×1 + 7×7×3+1+2 + 7 = 303	303

The calculated and the simulated values for the number of clock cycles used match each other but, in reality, because we have two wait states one is external evaluation wait state and one is the debugger wait state and because the clock count depends on other systems connected to GPeat, the total number of clock cycles may not be the same with calculated or simulated. The synthesis timing results are shown table 5.5. The number of clock cycles required is proportional to the number of chromosomes executed. We can observe from table 5.7 that the order of experiments with respect to total time taken for that experiment is experiment 1, experiment 4, experiment 2 and experiment 3 which is similar to order with respect to chromosomes executed experiment

1, experiment 4, experiment 2 and experiment 3 from table 5.2. Other EA parameters also effect time taken but very little compared to chromosomes executed.

5.6.2 Hardware Usage

Table 5.4 shows the total number of devices used, number of multipliers used, and delays for all four experiments. Based on the input parameters given in table 5.1 the hardware usage is analyzed.

Four clocks were used in experiment 1 and 4 but only three clocks are used in experiment 2 and 3. This is because experiments 1 and 4 use the debugger which requires an extra clock. Out of the three, one clock is used by main testbed and other two are used by components. Also, from the timing constraints table (Table 5.5) we can observe that the maximum output required time after clock (known as "hold time") for experiments 1 and 4 is 4.754ns and for experiments 2 and 3 is 4.040ns, a lesser value. This shows that GPeat timing also depends on debugger usage. We want to make an independent debugger that doesn't affect GPeat's workings. .

By observing total equivalent gate count for design of all experiments (from table 5.4), we didn't find any reasonable pattern between population and gate count (where experiment 2 has largest population size but experiment 3 has highest gate count) or total number of chromosomes and gate count (experiment 1 has highest number of chromosomes, in both defined and executed terms, but experiment 3 has highest gate count). None of the other parameters from experiment 1, experiment 2 and experiment 4 provide an explanation for the larger gate count for experiment 3. To investigate more, we took experiment 1 and experiment 3 parameters and exchanged them so as to check

their gate count in design summary reports. Parameters such as whether to use the debugger, population size, iterations and stopping criteria effects the number of gates used but we didn't observe any drastic or large difference in the gate count. But when it comes to reproduction type, we can observe a sizable change in number of devices used. This change in gate count is mainly due to change in flip-flops, logic related occupied slices and LUTs. The design summaries of experiment 1 and experiment 3 with parameter "OType" value exchanged is shown in Table 5.8. For comparison, the old design summaries of experiment1 and experiment3 from table 5.4 are also shown in Table 5.8.

Table 5.8 Design analysis table

		Actual number	**Old**		**New**	
			Exp 1	Exp 3	Exp 1	Exp 3
Logic utilization	Total Number Slice registers • used as Flip Flops • used as Latches	9,312	889 (9%) 856 33	1264 (13%) 1,250 14	1,498 (16%) 1,464 34	753 (8%) 739 14
	Number of 4 input LUTs	9,312	1,343 (14%)	1,559 (16%)	1,927 (20%)	1,186 (12%)
Logic Distribution	Number of occupied Slices • only related logic • unrelated logic	4,656	1,114 (23%) 1,114 0	1,416 (30%) 1,416 0	1,734 1,734 0	977 (20%) 977 0
	Total Number of 4 input LUTs • used as logic • used as a route-thru • used for	9,312	1,628 (17%) 1,343 253	1,799 (19%) 1,559 218	2,212 (23%) 1,927 253	1,426 (15%) 1,186 218

	Dual Port RAMs		32	22	32	22
	Number of bonded IOBs • IOB Flip Flops	232	81 (34%) 48	81(34%) 11	81(34%) 50	81(34%) 9
	Number of GCLKs	24	4 (16%)	3(12%)	4(16%)	3(12%)
	Number of MULT18X18SIOs	20	1 (5%)	1(5%)	1(5%)	1(5%)
	Total equivalent gate count for design		21,070	24,091	30,098	17,731
	Additional JTAG gate count for IOBs		3,888	3,888	3,888	3,888

From these values we can conclude that, even though we are using chromosomes of same length for all experiments, the total number of devices used mainly depends on the reproduction type (OType) while other parameters population size, iterations, debug and stopping criteria add to that. From this observation we can say that our crossover and mutation block have a lot of logic to be executed in one clock cycle which affects the performance of GPeat. In future versions, an improvement in the organization of the crossover and mutation blocks should, therefore, be looked at.

5.7 Comparison of Simulation and Synthesis Results.

To see if the simulation and hardware implementations match, simulation and synthesis results of experiment 2 and 3 have been compared. We used the debugger in experiments 2 and 3 in order to compare all our results. The tables 5.3 and 5.6 show all the chromosomes along with their fitness values, of simulation and synthesis respectively. By observing these values we can conclude that our simulation and synthesis result are same.

Graphs which show the fitness values for successive generations show that the chromosomes improve through the reproductive cycles. Graphs are plotted for experiments 1 and 4 with fitness on Y-axis and generations on X-axis. It is difficult to analyze experiment 2 and 3 since they met their termination criteria in 2 and 1 iterations respectively so the graphs for only experiment 1 and 4 are shown below. In these graphs chromosomes that are randomly generated are shown with red dots and chromosomes from previous solutions are shown with blue dots. An average fitness curve is also plotted in these graphs.

Graph1 (Fig 5.3) is drawn for fitness and generations of experiment 1. This graph is drawn from simulation and synthesis results which proved to be same of experiment 1. In this graph, X-axis range is (0-10) as the maximum number of iterations are 10 and the Y-axis range is (7-17) as the there are no fitness values out of range for this experiment.

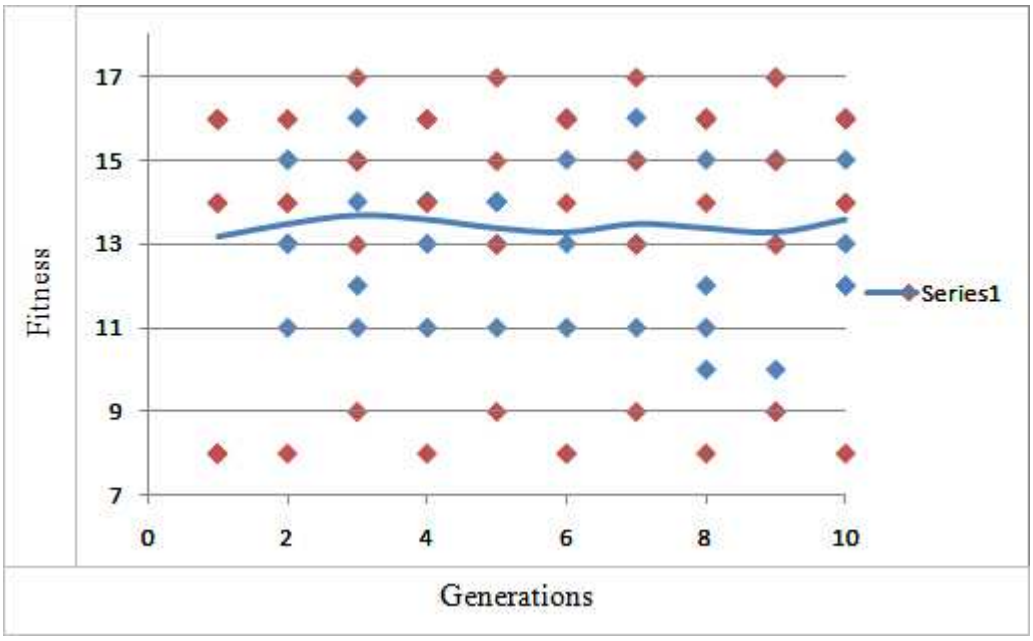

Figure 5.3 Graph between fitness and generations for experiment 1.

From graph 1 we can observe that most of the blue dots (fitness of chromosomes taken from previous solutions) are concentrated between 11 and 15 while red dots (fitness of randomly generated chromosomes) are distributed all through the range from 8 to 17. The ups and downs in the average fitness curve are because of the jumping nature of randomly generated chromosomes. From this we can say that by inducing random chromosomes in every generation we are forcing the chromosomes towards the global maxima instead of local maxima.

Graph 2 (Fig. 5.4) is drawn for fitness and generations of experiment 4. This graph is drawn from simulation and synthesis results (which proved to be same) of experiment 4. In this graph, X-axis range is (0-7) as the maximum number of iterations are 7 and the Y-axis range is (8-18) as the there are no fitness values out of this range for this experiment.

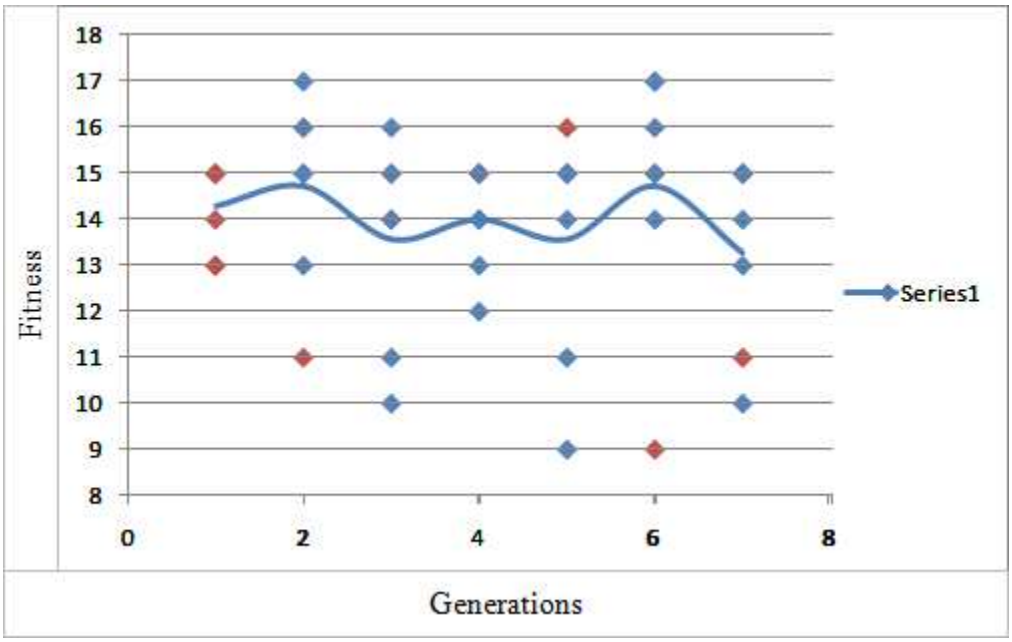

Figure 5.4 Graph between fitness and generations for experiment 4.

Here the elitism rate is very high and the results are less random. From graph 2 we can observe that most of the blue dots (fitness of chromosomes taken from previous solutions) are concentrated between 13 and 16 while red dots (fitness of randomly generated chromosomes) are distributed across the range from 9 to 16. The ups and downs in the average fitness curve are because of the jumping nature of randomly generated chromosomes. Even though there are few in number but shows good random nature leading to more fluctuations in the average curve. In this experiment target is not achieved. Looking at the graph we can say that adding some more random chromosomes and running for more generations leading to more fluctuations in the average cure which indeed force chromosomes towards global maxima target can be reached.

These comparisons help us in building a good future version of GPeat. From all these results and comparisons, we can say that our first version of GPeat system can be used for implementing a variety of EAs in hardware satisfactorily.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

Evolutionary algorithms have proven to be effective in addressing many real world problems like path optimization, prosthetic adjustments, design and testing of integrated circuits etc. [14-18]. Though these algorithms were applicable to problems in the physical world, many were tested in software in extrinsic runs where their fitness was determined by simulation tools. Though this allowed for investigation of EAs, many of the advantages of hardware were never exploited. We developed a reconfigurable, flexible system to make porting EA systems from software to hardware less challenging. This system lets the user enter system description parameters through a GUI and hides much of the hardware's details so that even those with minimal hardware experience can run their designs in the physical world. This general purpose evolutionary algorithm testbed system is called "GPeat".

GPeat allows quick implementation of different evolutionary algorithms in hardware. The GUI that GPeat provides is a friendly interface through which parameters for different algorithms can be specified. We developed a hardware architecture and VHDL code which is configured according to these parameters. GPeat also has a debugging interface which provides feedback from the system on individual chromosome

values and their fitness values among other things which helps user in analyzing the design's performance. In the next version of GPeat a breadboard area will be provided through which different sensors can be connect so information from the environment can be used to determine fitness values. Because the hardware knowledge required to use our GPeat system is minimal, GPeat reduces the barriers of evolutionary algorithm designer's to implement their designs in hardware.

An application that was implemented on the GPeat was a color theory application. Four experiments were conducted where the system was observed as it tried to find an unknown color, gold, red-orange, copper and blue-green. All these experiments were conducted with different sets of input parameters so as to test all the available options of GPeat. All those experiments were simulated in software as well as and synthesized and run on the hardware. The debugger allowed comparison of the simulated and synthesized results which proved to be same.

Thus we can conclude that a satisfactory first version of GPeat for quick implementation of evolutionary algorithms into hardware has been built.

6.2 Future Work

Upon completion of first version of GPeat, we began planning for future versions with improvements to make the system more efficient and effective. In the next version, we would like to upgrade the GUI while polishing each and every block of the hardware. We are also planning to add the sensor panel area of the system for next version and plan on adding more display options to the debugging system.

Run time and hardware usage are two important aspects of any hardware system. In the systems that we have looked at, the intrinsic and extrinsic evaluations are the slowest steps in the cycle. Ideally creating and running more of these evaluation blocks would give the biggest improvement in speed but in our present system we have no control of those blocks. In the next GPeat version we will offer multiple connections with intrinsic/extrinsic evaluation blocks so that more than one can be run in parallel. We will also offer the option for the user to provide their intrinsic test block as a VHDL component and, when GPeat is compiled and resource information becomes available, GPeat will be compiled a second time with as many of the intrinsic test blocks as possible implemented in parallel. Because the test blocks are the slowest piece of the system this will give the biggest performance improvements but, along with the test blocks, it will be looked at how to reduce looping code inside of our VHDL in order to reduce the synthesis time and use fewer resources in the GPeat code itself.

A small increase in the speed of GPeat can be realized by a change in processing techniques. For this first version of GPeat, the mutations and crossovers done on chromosomes are run done in series. By having more evaluation blocks in parallel we can run chromosomes in parallel which will decrease the run time of the system. It will also be worth looking at the chances of introducing parallelism in other blocks including the sorting, initial population generation and crossover and mutation blocks.

The introduction of pipelining to the GPeat system may also improve the speed of the system. In this first version, for example after sending a chromosome from reproduction block to fitness evaluation block, the reproduction block stays idle until first chromosome is fully processed. By introducing pipelining, the crossover block will

process second chromosome while first chromosome is under process in fitness evaluation block.

The hardware usage can be reduced by polishing certain blocks of GPeat. One main block that needs to be improved to save hardware is the storage block. In this version, we are using three bins of

$$3 \text{ bins} \times \text{population size} \times \text{iterations (generations)}$$

In future version, a single partitioned bin can replace those three bins. Before deciding on the approach to reduce memory there needs to be an evaluation of the tradeoffs. Moving all of the data to a single memory will most likely increase run-time. With the improvement in hardware structures we also hope to have a solid enough system that one day it may be implementable in a single ASIC (application specific integrated circuit).

In this first version the most commonly used crossover and mutation methods have been provided but in future versions we would like to provide more options and, if possible, a user definable option. Providing more crossover and mutation options as well as other system options will allow GPeat to support more kinds of EAs.

The sensors block is an area that needs implementation and is not available for this first version of GPeat. The sensor bank would be used in systems that run externally to the Xilinx board GPeat is run on. It could be used for simple interfacing with another digital circuit in serial or parallel or it could be used to actually gather information on the environment. An example where GPeat would just be “talking” with another system would be in the case where an autonomous car has all its sensors on the car and it sends back the data on obstacles as a digital signal already processed or even the actual fitness

value itself. An example of where the sensor bank would be gathering data from the environment is, if the GPeat system was put on a board and actually attached to the autonomous car and the sensors for collisions and angle of descent were directly attached to the board instead of the car. Then GPeat would gather the raw data, analyze it according to the rules entered by the user through the GUI and fitness would be generated internally are actually on the GPeat board simply need input and output from run in an environment. Sensor data may also be used in conjunction with the rules entered by the user to generate the initial population by helping in discarding illegal solutions in first step of iteration. This would lead to a population with more fit chromosomes for that environment. Sensors information can also be used in the rules for calculating fitness.

On GUI side, the main goal for next version is to provide options for the debugger that allows the drawing of graphs based on debugging information. These graphs would help users in analyzing a design's performance by making numerical data more friendly and easier to manipulate. For example, a graph which shows the fitness values for each iteration will show the improvement in chromosomes.

For future versions we would also like to implement learning abilities and better evaluation options including fuzzy methods and rules with probability attached.

The GPeat project has many topics yet to address but the initial successes have shown us that the theory is sound and that is a useful and flexible tool for quick generation and debugging of EA systems in hardware.

REFERENCES

1) Bentley, P.J., ed.: Evolutionary Design by Computers. Morgan Kaufmann (1999).

2) O'Neill, M., Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers (2003)

3) Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press (1992)

4) G. W. Greenwood, A. M. Tyrrell, "Introduction to Evolvable Hardware". IEEE Press Series on Computational Intelligence, New Jersey, USA. 2007

5) Mitchell, Melanie. "*An Introduction to Genetic Algorithms",* MIT Press, 1996

6) Goldberg, David. "*Genetic Algorithms in Search, Optimization, and Machine Learning".* Addison-Wesley, 1989

7) Haupt, Randy and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley & Sons, 1998

8) Thomas Komarek and Peter Pirsch," Array Architectures for Block Matching Algorithms", IEEE Transaction on Circuits and Systems, Vol. 36, No. 10, October 1989.

9) S. V. Hum, M. Okoniewski, R. J. Davies, An evolvable antenna platform based on reconfigurable reflect arrays. Proceedings of 2005 NASA/DoD Conference on Evolvable Hardware, June 29 - July 1, 2005, pp. 139 -146

10) P. Salek, J. Tarasiuk, K. Wierzbanowski, Application of Genetic Algorithms to Texture Analysis. Crystal Research and Technology, Volume 34, pp. 1073- 1079, 1999.

11) I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, T. Higuchi, An evolvable hardware chip and its application as a multi-function prosthetic hand controller. Innovative applications of artificial intelligence conference innovative applications of artificial intelligence, Orlando, Florida, United States, 1999, pp. 182 - 187.

12) M. Fatih Tasgetiren, P. N. Suganthan, P. Quan-Ke, L. Yun-Chia, A genetic algorithm for the generalized traveling salesman problem. IEEE Congress on Evolutionary Computation, 25-28 Sept. 2007 pp. 2382 - 2389

13) A. Carter, Design and Application of Genetic Algorithms for the Multiple Traveling Salesperson Assignment Problems. Dissertation submitted to the faculty of the Virginia Polytechnic Institute and State University, 2003.

14) F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, Optimizing Area Loss in Flat Glass Cutting. Second International Conference On Genetic Algorithms in Engineering Systems: Innovations and Applications, 2-4 Sep 1997 pp. 450 - 455.

15) G. B. Shelbe, K. Brittig, Refined genetic algorithm-economic dispatch example. IEEE Transactions on Power Systems, Volume 10, Issue 1, Feb. 1995 pp. 117 - 124.

16) M. A. Abido, A New Multiobjective Evolutionary Algorithm for Environmental/Economic Power Dispatch. IEEE Power Engineering Society Summer Meeting, 2001, Volume 2, Issue , 2001 Page(s):1263 - 1268 vol.2.

17) L. M. Garder, M. E. Hovin, Robot Gaits Evolved by Combining Genetic Algorithms and Binary Hill Climbing. Genetic algorithms: papers, 2006, pp. 1165 – 1170

18) E. Stomeo, T. Kalganova, C. Lambert, A Novel Genetic Algorithm for Evolvable Hardware. IEEE Congress on Evolutionary Computation, 2006, 16-21 July 2006 pp. 134 - 141

19) Stephen Coe, Shawki Areibi and Medhat Moussa, “A hardware Memetic accelerator for VLSI circuit partitioning”, 2007, Pergamon Press, Inc. Tarrytown, NY, USA

20) S.J. Flockton, K. Sheehan, “Evolvable hardware systems using programmable analogue devices,”’ IEEE Half-day Colloquium on EvolvableHardware Systems, Digest No. 1998/233, 3 March 1998 Page(s):5/1- 5/6

21) A. Stoica, R. Zebulum, D. Keymeulen, “Mixtrinsic Evolution,”’ Proc. of the 3rd Int. Conf. on Evolvable Systems, Edinburgh, U.K., 2000, pp. 208-217

22) J. Langeheine, S. Folling, K. Keir, J. Schemmel, “Towards a Silicon Primordial Sourp: A Fast Approach to Hardware Evolution with a VLSI Transistor Array,”’ Proc. Of the 3rd Int. Conf. on Evolvable Systems, Edinburgh, U.K., 2000, pp. 123-132

23) Fogel, D.B.: 'Evolutionary Computation: Toward a New Philosophy of Machine Intelligence' (IEEE Press, Piscataway, NJ, 1995)

24) Zebulum, R.S., Vellasco, M.M.R., and Pacheco, M.A.C.: 'Evolutionary Electronics: Automatic Design of Electronic Circuits' (CRC Press,Boca Raton, FL, 2001) Hollingworth, G., Smith, S., and Tyrrell, A.M.: 'Safe IntrinsicEvolution of Virtex

Devices'. Proc. 2nd NASA/DoD Workshop Evolvable hardware, Silicon Valley, CA, July 2000, pp. 195–202

25) Smilkstein, T.; Tati, K.K.; Barve, P.; Hai, M.L.; Sajjapongse, K.; Sharma, D.K. "An evolutionary algorithm testbed for quick implementation of algorithms in hardware", EDIS, 2009. IEEE Workshop on Volume, March 30 2009-April 2 2009 Page(s):51 - 57

26) Joan Boyar (1989). "Inferring sequences produced by pseudo-random number generators", journal of the ACM, 36 (1): 129–141.2

7) Sorting algorithms. (n.d.). Retrieved from the Wiki: http://en.wikipedia.org/wiki/Sorting_ algorithm

29) John Mac Taggart. (n.d.). *color theory*. Retrieved from http://www.artyfactory. com/color_theory/color_theory_terms_1.htm

30) Colors mixing chart. (n.d.) In sugarcraft online. Retrieved from http://www.sugarcraft .com/catalog/coloring/colormixingchart.htm

APPENDIX

Main code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.GPEAT_include.all;
use work.RS232_reference.all;
entity GPeat1 is
        generic(  N_G:        integer:=chrsize;
                  G_S:        integer:=GSize---GSize);
        port (      clk: in std_logic;
                  start: in std_logic;
                  CnM:       out std_logic_vector(2 downto 0);
                  C_I:       out integer;
                  Fitness:    out integer;
                  tx_intf          : out std_logic;
                  rx_intf          : in std_logic;
                  reset  : in std_logic;
                  Data_out:     out std_logic_vector((N_G*G_S-1) downto 0);
end GPeat1;

architecture Behavioral of GPeat1 is
        component example
        port(
                        clk:     in std_logic;
                        color:   in integer;
                        start1:  in std_logic;
                        chr:     in std_logic_vector(N_G*G_S-1 downto 0);
                        done_E:  out std_logic;
                        fit1:    out integer);
         end component;

        component GPD_controller is
        port (
                        tx_intf          : out std_logic;
                        busy             : out std_logic;
                        Chromosome  : in std_logic_vector((chrsize*Gsize)-1 downto 0);
                        start            : in std_logic;
                        rx_intf          : in std_logic;
                        clk              : in std_logic;
                        reset            : in std_logic);
        end component;
```

```
component debouncer is
port (
                Q               : out std_logic;
                D               : in std_logic;
                clk             : in std_logic;
                reset           : in std_logic
        );
end component;

                constant G_Ty:              std_logic:=Gtype;
                constant C_M:               integer:=OType;
                constant seed:              integer:=GSeed;
                constant INCREMENTOR:       integer:=GInc;
                constant MUL1:              integer:=GMul;
                constant MODM:              integer:=GMod;
                constant m:                 integer:=PSize;
                constant iterations:        integer:=Inumber;
                constant T_C:               integer:=CType;
                constant S_P:               integer:=AType;
                constant T_M:               integer:=MType;
                constant K:                 integer:=XSP;--  range 0 to (left-1);
                constant s:                 integer:=1;--  range 1 to (left-2);
                constant P:                 integer:=XEP;--  range 0 to (left-2);
                constant Sensor:            std_logic_vector(3 downto 0):="1010";
                constant AF:                integer:=AFitness;
                constant percentage:        integer:=ERate;
                constant singlestep:        std_logic:=RType;
                constant IrE:               std_logic:=FEType;
                constant MIN:               integer:=MaxFit;
                constant MAX:               integer:=MinFit;
                constant Debugger:          std_logic:=DEB;
                constant S_C:               std_logic:=SCriteria;--SCriteria;
                constant color:             integer:=1;
                constant FCG:               integer:=FCGN;
                constant FMG:               integer:=FMGN;
                constant Default_out: std_logic_vector((N_G*3-1) downto 0):=Doutput;
type   ram_1  is  array (1 to (m*iterations)) of std_logic_vector((G_S*N_G-1) downto 0);

signal A,B,D,fit1 :integer ;
signal state,nextstate : integer range 0 to 8;
signal counter1,counter2,fit : integer:=1;
signal RND,C_n_M,FV,Str,done1,DB,WWW,XXX,ZZZ,YYY,done_E,start1: std_logic;
signal chr:         std_logic_vector((G_S*N_G-1) downto 0);
signal start2,busy,reset_int : std_logic;
signal chr1: std_logic_vector((G_S*N_G-1) downto 0);
```

```
begin
WWW<='1' when counter1>=(m) else '0';
XXX<='1' when counter2>=((m)*iterations) else '0';
YYY<='1' when counter1>=(m) or fit>=AF else '0';--------for 2 chr at a time
counter1=((m/2)+1)
ZZZ<='1' when counter2>=((m)*iterations) or fit>=AF else '0';--------for 2 chr at a time
counter2=((m)*iterations)

debouncer0 : debouncer port map (reset_int, reset, clk, '0');
GPD0  : GPD_controller port map (tx_intf, busy, chr1, start2, rx_intf, clk, reset_int);
example_RYB : example port map (clk,color,start1,chr,done_E,fit1);

p1 : process(start,state,ZZZ,RND,C_n_M,FV,Str,DB,busy,done1,done_E)
begin
RND<='0';C_n_M<='0';FV<='0';Str<='0';DB<='0';done1<='0';

case state is
when 0 => if start='1' then
                nextstate<=1;
          else
                nextstate<=0;
          end if;

when 1 => RND<='1';
                nextstate<=2;

when 2 => C_n_M<='1';
           if IrE='1' then
                nextstate<=3;
           else
                nextstate<=4;
           end if;

when 3 => if done_E='1' then
                nextstate<=5;
          else
                nextstate<=3;
          end if;

when 4 => FV<='1';
                nextstate<=5;

when 5 => Str<='1';
          if debugger='1' then
                nextstate<=6;
```

```
                else if counter1=(m) then
                      nextstate<=8;
                else
                      nextstate<=1;
                end if;
                end if;

when 6 => if busy='1' then
                      nextstate<=6;
                 else
                      nextstate<=7;
                end if;

when 7 => DB<='1';
                 if counter1<(m) then
                      nextstate<=1;
                 else
                       nextstate<=8;
                 end if;

when 8 => done1<='1';
                if S_C='0' then
                if singlestep='1' and YYY='1'  then
                       nextstate<=0;
                   elsif singlestep='0' and ZZZ='1'  then
                      nextstate<=0;
                   else
                      nextstate<=1;
                   end if;
                else
                   if singlestep='1' and WWW='1'  then
                      nextstate<=0;
                   elsif singlestep='0' and XXX='1'  then
                      nextstate<=0;
                   else
                      nextstate<=1;
                end if;
                end if;
end case;
end process;


P2 : process(clk)

variable tmp_A,tmp_B,tmp_c,tmp_D,C_1,C_2,off_1,off_2,I:  std_logic_vector
((G_S*N_G-1) downto 0);
```

```
variable T: integer:=0;
variable F1,F2,F3,X:integer:=0;
variable A1,B1,D1,A2,B2,D2,ki,kir:integer:=0;
variable tmp_out: std_logic_vector((G_S*N_G-1) downto 0);
variable Fir_RA,Fir_RD: integer;
variable R: integer:=1;
variable U: integer;
variable R_A:integer:=seed;
variable R_D:integer;
variable CXM:integer;
variable crossover1,mutation1: std_logic;
variable kiran,kiran1: std_logic_vector((G_S-1) downto 0);
variable F_R_D,F_R_D1,F_R_D2,F_R_D3,F_R_D4,F_R_D5,F_R_D6: integer;
variable F_R_A: integer:=(seed);
variable tmp_ramA1,tmp_ramB1,tmp_ramC1:ram_1;
variable left: integer:=(G_S*N_G-1);
variable Fir_seed: integer:=4;
variable Fir_MUL1: integer:=5;
variable Fir_INCREMENTOR:integer:=1;
variable Fir_MODM: integer:=4;
variable fir_rng1,fir_rng2:integer;
variable k1,k2,k3:integer  range 1 to m*iterations:=1;------
variable mid8,mid9:integer;
variable Default,Crossover,Mutation,Exclusive:std_logic;


begin

if (clk'event and clk='1') then
state<=nextstate;
start2<='0';start1<='0';

if RND='1' then
if kir=0 or percentage=0 then

for i in 0 to (N_G-1) loop
R_D:= ((R_A*MUL1)+INCREMENTOR) mod MODM;
C_1((i+1)*G_S-1 downto (i*G_S))  :=conv_std_logic_vector(R_D,G_S);
R_A:=R_D;
end loop;

else

if R>=1 and R<=percentage then
```

```
for i in 0 to (N_G-1) loop
R_D:= ((R_A*MUL1)+INCREMENTOR) mod MODM;
C_1((i+1)*G_S-1 downto (i*G_S))  :=conv_std_logic_vector(R_D,G_S);
R_A:=R_D;
end loop;
R:=R+1;

else

if U<=(A1-A2) then
          C_1:=tmp_ramA1(K1);
          K1:=K1+1;
  U:=U+1;

elsif U>(A1-A2) and U<=((B1-B2)+(A1-A2)) then
 C_1:=tmp_ramB1(K2);
 K2:=K2+1;
 U:=U+1;

elsif U>((B1-B2)+(A1-A2)) and U<=((D1-D2)+(B1-B2)+(A1-A2)) then
 C_1:=tmp_ramC1(k3);
  K3:=K3+1;
 U:=U+1;

  end if;
  end if;
  end if;

 for i in 0 to (N_G-1) loop
R_D:= ((R_A*MUL1)+INCREMENTOR) mod MODM;
C_2((i+1)*G_S-1 downto (i*G_S))  :=conv_std_logic_vector(R_D,G_S);
R_A:=R_D;
end loop;

 end if;

 if C_n_M='1' then

 if C_M=0 then default:='1';   crossover:='0';  Mutation:='0';  exclusive:='0';
 elsif C_M=1  then default:='0';   crossover:='1';  Mutation:='1';  exclusive:='0';
 elsif C_M=2  then default:='0';   crossover:='1';  Mutation:='0';  exclusive:='0';
 elsif C_M=3  then default:='0';   crossover:='0';  Mutation:='1';  exclusive:='0';
 elsif C_M=4  then default:='0';   crossover:='0';  Mutation:='0';  exclusive:='1';
 elsif C_M=5  then default:='0';   crossover:='0';  Mutation:='0';  exclusive:='0';
```

```
 end if;

if default='0' then

if exclusive='1' then

 R_D:= ((R_A*MUL1)+INCREMENTOR) mod MODM;   R_A:=R_D;
if R_A rem 2 = 0 then crossover1:='1';
else mutation1:='1'; end if;
CXM:= ((R_A*MUL1)+INCREMENTOR) mod 4;
end if;

  if Crossover ='1' or crossover1='1' then
          if T_C =1 or CXM=0 then
tmp_A:= C_2((G_S*N_G-1) downto K+1) & C_1(K downto 0);

   else if T_C=2 or CXM=1  then
tmp_A:= C_1((G_S*N_G-1) downto P+1) & C_2(P downto K+1) & C_1(K downto 0);

   else if T_C=3 or CXM=2 then
For q in 0 to (N_G-1) loop
             if q rem 2 = 0 then
             tmp_A((G_S-1)*(q+1) downto G_S*q):=C_2((G_S-1)*(q+1) downto G_S*q);
             else
             tmp_A((G_S-1)*(q+1) downto G_S*q):=C_1((G_S-1)*(q+1) downto G_S*q);
             end if;
 end loop;

   else if T_C=4  or CXM=3 then
                    if S_P=1 then
                   for i in 0 to (N_G*G_S)-1 loop
                   tmp_A(i):= C_2(i) and C_1(i);
                   end loop;
   elsif S_P=2 then
                   for i in 0 to (N_G*G_S)-1 loop
                   tmp_A(i):= C_2(i) nand C_1(i);
                   end loop;
   elsif S_P=3 then
                   for i in 0 to (N_G*G_S)-1 loop
                   tmp_A(i):= C_2(i) xor C_1(i);
                   end loop;
elsif S_P=4 then
                 for i in 0 to (N_G*G_S)-1 loop
                 tmp_A(i):= C_2(i) or C_1(i);
                 end loop;
```

```
  elsif S_P=5 then
                for i in 0 to (N_G*G_S)-1 loop
                tmp_A(i):= C_2(i) nor C_1(i);
                end loop;
  elsif S_P=6 then
               for i in 0 to (N_G*G_S)-1 loop
               tmp_A(i):= C_2(i) xnor C_1(i);
               end loop;
end if;
else if T_C=5 or CXM=4 then

for i in 1 to FCG loop----------F_R_D loop
 F_R_D1:= ((F_R_A*MUL1)+INCREMENTOR) mod 8;   F_R_A:=F_R_D1;

 F_R_D2:= ((F_R_A*MUL1)+INCREMENTOR) mod 8;   F_R_A:=F_R_D2;

kiran:=C_1((((F_R_D1+1)*G_S)-1) downto ((F_R_D1)*G_S));
kiran1:=C_2((((F_R_D2+1)*G_S)-1) downto ((F_R_D2)*G_S));

C_1((((F_R_D1+1)*G_S)-1) downto ((F_R_D1)*G_S)):=kiran1;
C_2((((F_R_D2+1)*G_S)-1) downto ((F_R_D2)*G_S)):=kiran;

end loop;
tmp_A:=C_1;
 end if;
end if;
end if;
end if;
end if;

else if crossover='0' then
          tmp_A:=C_1;
 end if;
 end if;
 if mutation='1' or mutation1='1' then
                if T_M=1  or CXM=0 then
                for i in 0 to (N_G*G_S)-1 loop
                tmp_C(i):= not tmp_A(i);
                end loop;
     else if T_M=2 or CXM=1 then
                For  q in 0 to (((G_S*N_G)/2)-1) loop
                          t:=2*q;
                          tmp_C(t):= tmp_A(t+1);
                          tmp_C(t+1):= tmp_A(t);
                end loop;
```

```
                if ((G_S*N_G)/2)=1 then
                tmp_C((G_S*N_G)-1):=tmp_A((G_S*N_G)-1);
                end if;

    else if T_M=3 or CXM=2 then
                tmp_C:=tmp_A(((G_S*N_G-1)/2) downto 0) & tmp_A((G_S*N_G-1)
                downto ((G_S*N_G-1)/2)+1);

    else if T_M=4  or CXM=3 then
                F_R_D3:= ((F_R_A*MUL1)+INCREMENTOR) mod 8;
                F_R_A:=F_R_D3;
      for i in 0 to G_S-1 loop
                tmp_A((F_R_D3+1)*G_S+i):= not tmp_A((F_R_D3+1)*G_S+i);
      end loop;
    tmp_C:= tmp_A;

else if T_M=5 or CXM=4 then

for i in 1 to FMG loop
 F_R_D4:= ((F_R_A*MUL1)+INCREMENTOR) mod 8;   F_R_A:=F_R_D4;

 if F_R_D4 rem 2=0 then
 F_R_D5:= ((F_R_A*MUL1)+INCREMENTOR) mod 127;   F_R_A:=F_R_D5;
  F_R_D6:= conv_integer(tmp_A((F_R_D4+1)*G_S-1 downto F_R_D4*G_S));
 tmp_A((F_R_D4+1)*G_S-1 downto
F_R_D4*G_S):=conv_std_logic_vector(F_R_D5*F_R_D6,G_S);
 --tmp_B((F_R_D4+1)*G_S-1 downto
F_R_D4*G_S):=conv_signed(F_R_D5*F_R_D6,G_S);

 tmp_C:=tmp_A;
 else

 F_R_D6:= conv_integer(tmp_A((F_R_D4+1)*G_S-1 downto F_R_D4*G_S));
  tmp_A((F_R_D4+1)*G_S-1 downto F_R_D4*G_S):=conv_std_logic_vector(-
1*F_R_D6,G_S);
 tmp_C:=tmp_A;
end if;
end loop;
end if;
end if;
end if;
end if;
end if;
  else if Mutation='0' then
            tmp_C:=tmp_A;
  end if;
```

```
 end if;
 else
 if default='1' then
  For q in 0 to (N_G-1) loop
            if q rem 2 = 0 then
            tmp_A((G_S*(q+1)-1) downto G_S*q):=C_2((G_S*(q+1)-1) downto G_S*q);
            else
            tmp_A((G_S*(q+1)-1) downto G_S*q):=C_1((G_S*(q+1)-1) downto G_S*q);
            end if;
 end loop;
for i in 0 to (N_G*G_S)-1 loop
            tmp_C(i):= not tmp_A(i);
            end loop;

 end if;
 end if;

off_1:=tmp_C;
if IrE='1' then
chr<=off_1;
start1<='1';
end if;
end if;

if FV='1' then
          if   off_1(0) = '1' and sensor(0)='1' then f1:= 100;
          elsif     off_1(1) = '1' and sensor(1)='1' then f1:=99;
          elsif     off_1(2) = '0' and sensor(2)='1' then f1:=82;
          elsif     off_1(3) = '0' and sensor(3)='1' then f1:=78;
          else      F1:= 38;
          end if;
end if;
if Str='1' then

if IrE='1' then
F1:=fit1;
end if;
          if  F1>=F3 then
          tmp_out:=off_1;
          F3:=F1;
          end if;
          mid8:=min+((max-min)/10)*6;
          mid9:=min+((max-min)/10)*8;
           if F1>=MIN and F1<mid8 then
                    D1:=D1+1;
```

```
        tmp_ramC1(D1):=off_1;
          else if F1>=mid8 and F1<mid9 then
          B1:=B1+1;
          tmp_ramB1(B1):=off_1;
          else if F1>=mid9  then
          A1:=A1+1;
        tmp_ramA1(A1):=off_1;
end if;
end if;
end if;

if Debugger='0' then
counter1<=counter1+1;
counter2<=counter2+1;
end if;

Fit<=f1;
end if;
if DB='1' then

start2<='1';
chr1<=std_logic_vector(off_1);
Fitness<=f1;
C_I<=counter2;
if C_M=4 then
CnM<=conv_std_logic_vector(CXM,3);
else
CnM<=conv_std_logic_vector(C_M,3);
end if;
counter1<=counter1+1;
counter2<=counter2+1;
end if;

if done1='1' then
start2<='0';
U:=1;ki:=0;R:=1;A2:=A1;B2:=B1;D2:=D1;counter1<=1;kir:=kir+1;
if singlestep='1' and  YYY='1' then
       if F3>=AF then
       Data_out<=tmp_out;
       else
       Data_out<=Default_out;
       end if;
elsif singlestep='0' and ZZZ='1' then
       if F3>=AF then
       Data_out<=tmp_out;
else
```

```
        Data_out<=Default_out;
end if;
end if;

A<=1000-A1;
B<=1000-B1;
D<=1000-D1;
end if;
end if;

end process;

end Behavioral;
```

Code for debouncer component

```
library ieee;
use ieee.std_logic_1164.all;
entity debouncer is
	port (		Q		: out std_logic;
			D		: in std_logic;
			clk		: in std_logic;
			reset		: in std_logic);
end entity;
architecture rtl of debouncer is
	constant cnt_hit, cnt_lot : integer range 0 to 50000000 := 6250000;
	signal cnt_hi, cnt_lo	: integer range 0 to 50000000;
	signal Qbuf	: std_logic;
begin
	Q <= Qbuf;
	process(reset, clk)
	begin
		if (reset = '1') then
			cnt_hi <= 0;
			cnt_lo <= 0;
			Qbuf <= '0';
		else
			if ((clk'event) and (clk = '1')) then
				if  (D = '1') then
					if (cnt_hi < cnt_hit) then
						cnt_lo <= 0;
						cnt_hi <= cnt_hi + 1;
					else
						cnt_hi <= 0;
						Qbuf <= '1';
					end if;
				elsif (D = '0') then
					if (cnt_lo < cnt_lot) then
						cnt_hi <= 0;
						cnt_lo <= cnt_lo + 1;
					else
						cnt_lo <= 0;
						Qbuf <= '0';
					end if;
				else
				end if;
			end if;
		end if;
	end process;
end rtl;
```

Code for GPD controller component

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.RS232_reference.all;
use work.GPEAT_include.all;

entity GPD_controller is
	port (
			tx_intf		: out std_logic;
			busy		: out std_logic;

			Chromosome	: in std_logic_vector((chrsize*Gsize)-1 downto 0);
			start		: in std_logic;
			rx_intf		: in std_logic;
			clk			: in std_logic;
			reset		: in std_logic
		);
end GPD_controller;

architecture Behavioral of GPD_controller is
	component Memory is
	port (
			Dout		: out std_logic_vector(Bitdepth-1 downto 0);

			Chromosome	: in std_logic_vector((chrsize*Gsize)-1 downto 0);
			LEN			: in std_logic;
			ADDR		: in std_logic_vector(7 downto 0);
			DStrb		: in std_logic;
			reset		: in std_logic
		);
	end component;
	component interface_controller is
	port (
			tx_intf : out std_logic;
			txFull  : out std_logic;
			txFin   : out std_logic;
			debug1: out std_logic;
			debug2: out std_logic;
			debug3: out std_logic_vector(1 downto 0);

			Din		: in std_logic_vector(Bitdepth-1 downto 0);
			mode   : in std_logic;
			Sin    : in std_logic;
```

```
                Send    : in std_logic;
                clk             : in std_logic;
                reset   : in std_logic
        );
    end component;
    component rx_interface_controller is
    port (
                Dout    : out std_logic_vector(Bitdepth-1 downto 0);
                D_avail         : out std_logic;
                Full    : out std_logic;

                rx_intf : in std_logic;
                mode    : in std_logic;
                Dstrb   : in std_logic;
                clk             : in std_logic;
                reset   : in std_logic
        );
    end component;

    signal mem_Dout      : std_logic_vector(Bitdepth-1 downto 0);
    signal mem_Chromosome : std_logic_vector((chrsize*Gsize)-1 downto 0);
    signal mem_ADDR    : std_logic_vector(7 downto 0);
    signal mem_LEN, mem_Dstrb : std_logic;

    signal tx_txFull, tx_txFin, tx_debug1, tx_debug2     : std_logic;
    signal tx_debug3       : std_logic_vector(1 downto 0);
    signal tx_Din  : std_logic_vector(Bitdepth-1 downto 0);
    signal tx_mode, tx_Sin, tx_Send : std_logic;

    signal rx_Dout: std_logic_vector(Bitdepth-1 downto 0);
    signal rx_D_avail, rx_Full : std_logic;
    signal rx_mode, rx_Dstrb : std_logic;

    type GPDC_states is (WAIT_CMD, FETCH_CMD, INST_DEC ,WAIT_START,
FETCH_DATA, FETCH_INFO, SEND);
    type FETCH_CMD_states is (s0,s1,s2);
    type FETCH_DATA_states is (s0,s1,s2,s3,s4,s5,s6,s7);

    signal state : GPDC_states;
    signal FETCH_CMD_state : FETCH_CMD_states;
    signal FETCH_DATA_state : FETCH_DATA_states;
    signal start_buf, start_toggle0, start_toggle1 : std_logic;
    signal PS_reg, BI_reg : std_logic_vector(Bitdepth-1 downto 0);
    signal busy_buf : std_logic;

begin
```

```
	Memory0 : Memory port map (mem_Dout, mem_Chromosome, mem_LEN,
mem_ADDR, mem_DStrb, reset);
	TX0     : interface_controller port map (tx_intf, tx_txFull, tx_txFin, tx_debug1,
tx_debug2, tx_debug3, tx_Din, tx_mode, tx_Sin, tx_Send, clk, reset);
	RX0 : rx_interface_controller port map (rx_Dout, rx_D_avail, rx_Full, rx_intf,
rx_mode, rx_DStrb, clk, reset);

	busy <= busy_buf;
	mem_Chromosome <= Chromosome;

	process(clk, reset)
	begin
		if (reset = '1') then
			state <= WAIT_CMD;
			start_toggle0 <= '0';
			FETCH_CMD_state <= s0;
			FETCH_DATA_state <= s0;
		else
			if (clk'event and clk='1') then
			busy_buf <= GPD_controller_busy;

			tx_mode <= interface_controller_buffering;
			tx_Sin <= '0';

			rx_mode <= rx_interface_controller_buffering;
			rx_DStrb <= '0';

			mem_LEN <= '1';
			mem_ADDR <= "00000000";
			mem_DStrb <= '0';

			case state is
----------------------------------WAIT_CMD-----------------------------------------------------
			when WAIT_CMD =>
			--dstate <= 0; --debug
			if (rx_D_avail = '1') then
			state <= FETCH_CMD;
			else
			state <= WAIT_CMD;
			end if;
----------------------------------FETCH_CMD----------------------------------------------------
			when FETCH_CMD =>
			--dstate <= 1; --debug
			case FETCH_CMD_state is
			when s0 =>
			--dFETCH_CMD_state <= 0; --debug
```

```
                        rx_mode <= rx_interface_controller_receiving;
                        rx_DStrb <= '0';
                        FETCH_CMD_state <= s1;
                        state <= FETCH_CMD;
                        when s1 =>
                        --dFETCH_CMD_state <= 1; --debug
                        rx_mode <= rx_interface_controller_receiving;
                        rx_DStrb <= '1';
                        FETCH_CMD_state <= s2;
                        state <= FETCH_CMD;
                        when s2 =>
                        --dFETCH_CMD_state <= 2; --debug
                        rx_mode <= rx_interface_controller_receiving;
                        rx_DStrb <= '0';
                        FETCH_CMD_state <= s0;
                        state <= INST_DEC;
                        end case;
----------------------------------WAIT_START-----------------------------------------------------

                        when INST_DEC =>
                        --dstate <= 2; --debug
                        case rx_Dout is
                        when "01110010" =>
                        state <= WAIT_START;
                        when "10101010" =>
                        state <= FETCH_INFO;
                        when "00001111" =>
                        state <= FETCH_INFO;
                        when others =>
                        state <= WAIT_CMD;
                        end case;
----------------------------------WAIT_START-----------------------------------------------------

                        when WAIT_START =>
                        --dstate <= 3; --debug
                        if (start_buf = '1') then
                        state <= FETCH_DATA;
                        start_toggle0 <= not start_toggle0;
                        else
                        state <= WAIT_START;
                        mem_LEN <= '0';
                        end if;
                        busy_buf <= GPD_controller_available;
----------------------------------FETCH_DATA-----------------------------------------------------
                        when FETCH_DATA =>
                        --dstate <= 4; --debug
```

```
mem_LEN <= '0';
case FETCH_DATA_state is
when s0 =>      --Prepare to fetch Package size
--dFETCH_DATA_state <= 0; --debug
mem_ADDR <= "00000000";
mem_Dstrb <= '1';
tx_Sin <= '0';
FETCH_DATA_state <= s1;
when s1 => --Fetch Package size
--dFETCH_DATA_state <= 1; --debug
mem_ADDR <= "00000000";
mem_Dstrb <= '0';
PS_reg <= unsigned(mem_Dout) + Infopacks;
BI_reg <= "00000000";
tx_Sin <= '0';
FETCH_DATA_state <= s2;
when s2 => --Increae BI_reg by 1
--dFETCH_DATA_state <= 2; --debug
BI_reg <= unsigned(BI_reg) + 1;
mem_Dstrb <= '0';
tx_Sin <= '0';
FETCH_DATA_state <= s3;
when s3 => --Prepare to Fetch byte from RAM
--dFETCH_DATA_state <= 3; --debug
mem_ADDR <= BI_reg;
mem_Dstrb <= '0';
tx_Sin <= '0';
FETCH_DATA_state <= s4;
when s4 => --Fetch byte from RAM
--dFETCH_DATA_state <= 4; --debug
mem_ADDR <= BI_reg;
mem_Dstrb <= '1';
tx_Sin <= '0';
FETCH_DATA_state <= s5;
when s5 => --Prepare to Push byte to FIFO
--dFETCH_DATA_state <= 5; --debug
mem_ADDR <= BI_reg;
mem_Dstrb <= '0';
tx_Din <= mem_Dout;
tx_Sin <= '0';
TCH_DATA_state <= s6;
hen s6 => --Push byte to FIFO
--dFETCH_DATA_state <= 6; --debug
mem_ADDR <= BI_reg;
mem_Dstrb <= '0';
tx_Din <= mem_Dout;
```

```
                        tx_Sin <= '1';
                        FETCH_DATA_state <= s7;
                        when s7 => --Finish pushing and prepare to send
                        --dFETCH_DATA_state <= 7; --debug
                        mem_ADDR <= BI_reg;
                        mem_Dstrb <= '0';
                        tx_Din <= mem_Dout;
                        tx_Sin <= '0';
                        if (BI_reg < PS_reg) then
                        FETCH_DATA_state <= s2;
                        else
                        FETCH_DATA_state <= s0;
                        state <= SEND;
                        end if;
                        end case;
-----------------------------------FETCH_INFO-----------------------------------------------------

                        when FETCH_INFO =>
                        --dstate <= 5; --debug
                        when SEND =>
                        --dstate <= 6; --debug
                        tx_mode <= interface_controller_sending;
                        if (tx_txFin = '1') then
                        state <= WAIT_CMD;
                        else
                        state <= SEND;
                        end if;
                        end case;
                        end if;
                end if;
        end process;

        start_buf <= start_toggle0 xor start_toggle1;

        process(reset, start)
        begin
                if (reset = '1') then
                        start_toggle1 <= '0';
                else
                        if ((start'event) and (start = '1')) then
                                start_toggle1 <= not start_toggle1;
                        end if;
                end if;
        end process;

end Behavioral;
```

Code for example_RYB component

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity example is
generic( N_G: integer:=3;
                        G_S: integer:=3;
                        MF: integer:=21);
port(
                clk:    in std_logic;
                color:  in integer;
                start1: in std_logic;
                chr:    in std_logic_vector(N_G*G_S-1 downto 0);
                done_E: out std_logic;
                fit1:   out integer);
end example;
architecture Behavioral of example is
begin
process(clk)
variable Fitness1: integer:=0;
variable RYB1,RYB2,RYB3: integer;
begin
if start1='1'then
RYB1:= conv_integer(chr((N_G*G_S-1) downto (N_G*(G_S-1))));
RYB2:= conv_integer(chr((N_G*(G_S-1)-1) downto (N_G*(G_S-2))));
RYB3:= conv_integer(chr((N_G*(G_S-2)-1) downto (N_G*(G_S-3))));
if color=1 then
        fitness1:= (MF-(abs(3-RYB1)+ abs(6-RYB2)+abs(1-RYB3)));
        elsif color=2 then
        fitness1:= (MF-(abs(3-RYB1)+ abs(2-RYB2)+abs(0-RYB3)));
        elsif color=3 then
        fitness1:= (MF-(abs(1-RYB1)+ abs(2-RYB2)+abs(1-RYB3)));

        elsif color=4 then
        fitness1:= (MF-(abs(0-RYB1)+ abs(1-RYB2)+abs(2-RYB3)));
        end if;

done_E<='1';
fit1<=fitness1;
end if;
end process;
end Behavioral;
```

Code for including RS-232 package

```
library ieee;
use ieee.std_logic_1164.all;
use work.GPEAT_include.all;

package RS232_reference is

	constant BitDepth : integer := 8;
	constant datapacks : integer := (chrsize*Gsize) / Bitdepth;
	constant dataremain : integer := (chrsize*Gsize) mod Bitdepth;
	constant Infopacks : integer := 2;
	constant BufferSize : integer := datapacks+1+Infopacks;
	constant BaudHighTime : integer := 54;--2604;
	constant BaudLowTime : integer := 54;--2604;
	constant StartBit : std_logic := '0';
	constant StopBit : std_logic := '1';
	constant interface_controller_buffering : std_logic := '0';
	constant interface_controller_sending :std_logic := '1';
	constant rx_interface_controller_buffering : std_logic := '0';
	constant rx_interface_controller_receiving : std_logic := '1';
	constant GPD_controller_busy : std_logic := '1';
	constant GPD_controller_available : std_logic := '0';
	type memory is array (0 to BufferSize-1) of std_logic_vector(BitDepth-1 downto 0);

	function repeat(N: natural; B: std_logic) return std_logic_vector;

end RS232_reference;

package body RS232_reference is
	function repeat(N: natural; B: std_logic) return std_logic_vector is
		variable result: std_logic_vector(1 to N);
	begin
		for i in 1 to N loop
			result(i) := B;
		end loop;
		return result;
	end;
end RS232_reference;
```

GPeat include file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package GPEAT_include is
	constant chrsize : integer := 3; --The size of Chromosomes
	constant GType : std_logic := '1'; --Gene Type ('0'-binary, '1'-integer)
	constant SCriteria : std_logic := '0'; --Stopping Criteria Type ('0'-iteration, '1'-fitness)
	constant OType : integer :=5; --Operation Type (0-default, 1-both, 2-Xover only, 3-
Mutation only, 4-Exclusive, 5-None)
	constant GSize : integer := 3; --Gene Size
	constant GSeed : integer :=6; --Gene Seed
	constant GInc : integer := 3; --Gene Incrementor
	constant GMul : integer := 5; --Gene Multiplier
	constant GMod : integer := 8; --Gene Modulus
	constant PSize : integer := 10; --Population Size
	constant INumber : integer :=10; --Maximum Iteration number
	constant CType : integer := 4; --Crossover type (1-Single, 2-Two, 3-Uniform, 4-
Arithmetic, 5-FIR)
	constant FCGN : integer := 2; --FIR-type Gene number for crossover
	constant FMGN : integer := 2; --FIR-type Gene number for mutation
	constant AType : integer := 1; --Arithmetic Crossover type (1-AND, 2-OR, 3-NAND, 4-
NOR, 5-XOR, 6-XNOR)
	constant MType : integer := 3; --Mutation type (1-Flip gene, 2-Exchange adjacent gene,
3-Mirror, 4-Randomly flip, 5-FIR)
	constant XSP : integer := 0; --Crossover Starting-point
	constant XEP : integer := 1; --Crossover Ending-point
	constant AFitness : integer := 17; --Acceptable Fitness Value
	constant MaxFit: integer := 21; --Maximum Fitness Value
	constant MinFit : integer := 0; --Minimum Fitness Value
	constant DOutput : std_logic_vector := "000000000"; --Default Output
	constant ERate : integer := 5; --Elitism rate
	constant FEType : std_logic := '1'; --Fitness Evaluation type ('0'-internal, '1'-
intrinsic/extrinsic)
	constant RType : std_logic := '0'; --Run type ('0' Continuous, '1' Single-step)
	constant DEB : std_logic := '1'; --Debuger Enable

end GPEAT_include;

package body GPEAT_include is
end GPEAT_include;
```