GUI FOR DEBUGGING AND DEVELOPMENT OF RECONFIGURABLE, EVOLVING SYSTEMS ENABLED BY EMBEDDED HARDWARE BLOCKS

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

KITTISAK SAJJAPONGSE

Dr. Tina Smilkstein, Advisor

MAY 2010

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

GUI FOR DEBUGGING AND DEVELOPMENT OF RECONFIGURABLE, EVOLVING SYSTEMS ENABLED BY EMBEDDED HARDWARE BLOCKS

presented by **Kittisak Sajjapongse**

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. James M. Keller, Professor, Department of Electrical and Computer Engineering

Dr. Yunxin Zhao, Professor, Department of Computer Science

ACKNOWLEDGEMENTS

I would like to thank Dr. Tina Smilkstein for her assistance and guidance through out my research. I would also like to thank Dr. James Keller and Dr. Yunxin Zhao for therir valuable suggestions and for serving on my committee.

I wish to thank my research mates, Kiran Kumar Tati and Parashar Barve, for their comments, help preparing this paper and friendship.

I would especially thank to my parents, Dr. Adisak Sajjapongse and Tommarat Sajjapongse, and my elder sisters, Nok Jurairat Sajjapongse and Mam Naovarat Sajjapongse, for their love, care and support.

I am in deep gratitude and admiration to Parn Sasiwimon Yoo-eam who has always kept me optimistic and stayed beside me all the time.

I take this opportunity to thank my friends, Rocky Surachat Pumipuntu, Frank Surachai Pumipuntu, Prite Chaowalit Thamsonglar, Ju Kanchana Songkittiphong and all my friends who stood by my side for their suggestions and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GENERAL PURPOSE EVOLUTIONARY ALGORITHM TESTBED

Kittisak Sajjapongse

Dr. Tina Smilkstein, Advisor

ABSTRACT

A flexible hardware testbed for Evolutionary Computation is developed in our research. The purpose is to realize a system with an ability to implement a wide variety of Evolutionary Computation applications. In this literature, General Purpose Evolutionary Algorithm Testbed (GPEAT) is described. GPEAT provides a flexible substrate for Evolutionary Computation applications on hardware platform using reconfigurable device such as FPGA. GPEAT consists of GPEAT core, debugging hardware, configuring GUI (Parameter-entry GUI) and debugging GUI. Parameter-entry GUI enables users to construct an Evolutionary Computation system on hardware with minimal knowledge on hardware designing. The GPEAT core obtains system parameters through the GUI and operates accordingly as users specify to find appropriate solutions from search space. Debugging interface is also provided through debugging hardware and debugging GUI to assist users in evaluating implemented system.

GPEAT provides users friendly environment to instantiate and debug Evolutionary Computation application on hardware. VHDL code is generated according to the system description parameters by the GUI. The design, then, be programmed into reconfigurable device. Xilinx ISE and ModelSim XE-III (MXE-III) simulator are used as tools for implementing the GPEAT system on Xilinx® Spartan 3E™ starter kit.

Chromosomes, fitness values and other information of the implemented system is pulled out from the device through debugging hardware, which is attached to the GPEAT core, and is visualized by debugging GUI. 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.

CHAPTER 1

INTRODUCTION

1.1 Introduction

Evolutionary Computation (EC) is a method for finding a set of appropriate solutions to optimization problems. It is a heuristic search process which imitates the process of gradual development found in natural habitats. Living organisms evolve through the interaction of competition, selection, reproduction and mutation processes. EC has been used to solve many kinds of real world problems. It provides efficient search and optimization tools for very large dataset and does not require mathematical modeling. The effectiveness of ECs include their high suitability for parallel computer implementation, particular success in large search and optimization problems and an their ability to learn complex relationships in incomplete datasets [1]. Mathematicians and computer scientists have been exploring the possibilities of ECs for many years [2-4].

Running ECs on software have a number of disadvantages over running them on hardware. These include slower execution speed when emulated by software on a conventional computer and the need to build a dedicated platform for mobile device or embedded system which may not be able to provide sufficient mobility or memory to execute programs that require a large dataset. Also, for mathematicians and computer scientists, the idea of moving EC into hardware may be too much of a challenge due to the required need of hardware knowledge. Implementing an EC system on hardware seems to be painful since each building block in the system must be customized for one set of parameters in a particular optimization problem. Changing parameters could mean changing the hardware structure of a building block which requires many design considerations including debugging the hardware block, design time and how it interfaces with other blocks. Once the hardware is built, debugging and performance analysis tools are needed. Each time a new system is built a new way to debug and evaluate must also be developed. To enable mathematicians and scientists who need to yield the benefits of hardware parallelism, we have designed a reconfigurable EC testbed (General Purpose Evolutionary Algorithm Testbed, GPEAT) to provide an easy platform to design and debug hardware EC systems.

GPEAT is a hardware framework generalized for EC application. Common hardware structures for EC are used to enable efficient execution. Field-Programmable-Gate-Array (FPGA) is used as a substrate for these hardware structures. In the design of GPEAT, not only was design effort put into the development of hardware structures, much effort was also put into making the user interface specific to the development of EC systems. GPEAT can be reconfigured and debugged through Graphical-User-Interface (GUI) provided as a configuring and debugging tool. In this way we developed GPEAT which can be used to design and run a wide variety of EC application as well as allowing a person with minimal hardware knowledge to design EC systems in hardware.

1.2 Review of Evolutionary Computation

Evolutionary Computations are search techniques that are biologically inspired. The techniques search for qualified solutions, or *candidates*, in a search space for particular problems. The search space is an abstract surface which can have as many dimensions as required by the problem. EC search for a region or a point in the search space in which local-minima (-maxima) or global-minima (-maxima) is located. A set of candidates, namely a *population*, represents a specific region of the search space. *Fitness value* is introduced to evaluate and categorize candidates and used to represent attribute for a point in the search space. Fitness value can be calculated by a function which is defined by the problem to be solved or optimized. Each candidate can be represented in a bio-inspired form called *chromosome*; and each chromosome consists of a number of *genes*.

An example of search space is shown in Figure 1.1, the entire search space for 2-dimension (2-gene) problem is presented with many local-minima (-maxima) and one global-maxima. A population can represents any specific region of the surface; and if the population is large enough, it virtually represents the entire search space. If this is the case, the technique search for a peak point which can be any local-maxima or the global-maxima for optimizing the problem. The search space in Figure 2.1 can be depicted because it is generated from a deterministic sinc function. In real world application, search space might be unable to be depicted due to non-deterministic characteristic of the problem. A problem can have more than two dimensions which can make the solutionspace harder to describe. Sometimes, we only know how to evaluate each individual candidate so it is impossible to construct the entire search space but, because a number of candidates representing the entire search space can be uncounted, the problem becomes tractable.

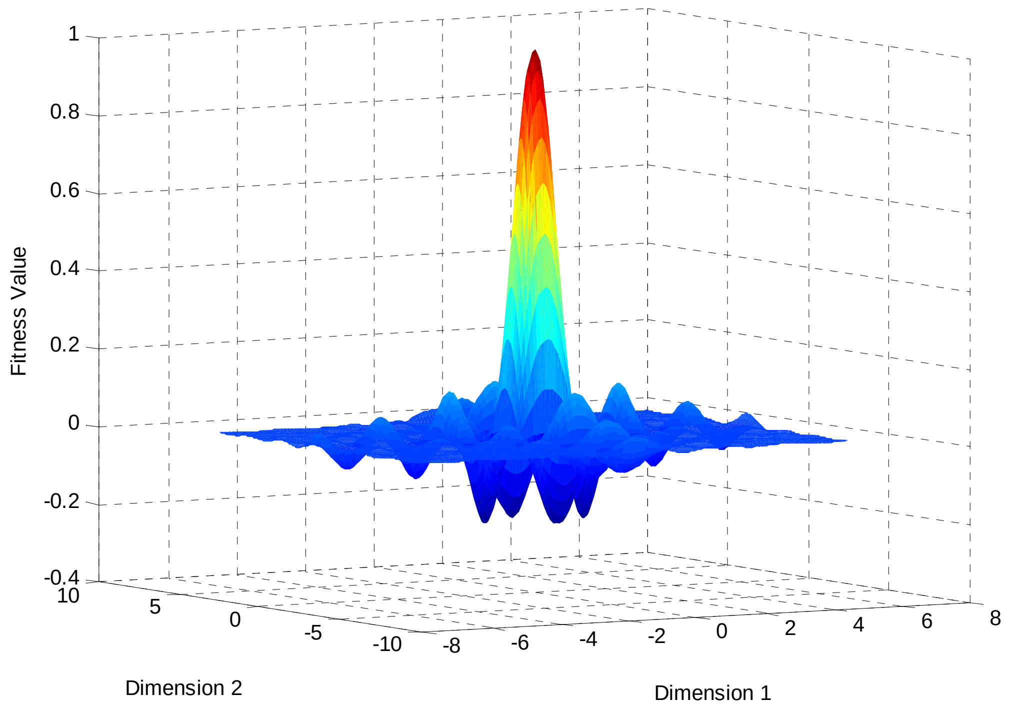


Figure 1.1 an example of Search space

1.2.1 Chromosome and Gene Representation

In a population, each candidate has some information encoded in it in some fashion. This information must be appropriate in order to solve or optimize a problem. A group of blocks containing numeric or symbolic data are combined to form a single candidate which can be interpreted as a single possible solution to a particular problem. This resembles to biological chromosome which contains sequences of genes. For this reason, we call a single block containing data as a *gene* and a candidate a *chromosome* in EC. A Basic Structure for a chromosome is shown in Figure 1.2.

Gene 1	Gene 2	Gene 3	…………….	Gene N

Figure 1.2 Chromosome structure

The commonly used encoding or value of each block include binary strings, integer numbers, fixed-point numbers, floating-point numbers, graphs and hybrid values. This means each gene can contain a binary bit-string, a number, or some combination of symbols. Genes in a chromosome are considered as a unit and are evaluated as a solution as such. The quality or fitness of a chromosome as a solution to a problem is evaluated by looking at how, when the gene values are used as a group as the input values to a problem, how well the problem is solved. This evaluation of how good a solution a chromosome is can be donr by using a set of rules or by comparing performance from a test run in an actual environment or simulation. Chromosome evaluation will be described in section 1.1.4.

A popular example for EC will be used here to illustrate. Suppose that there is a salesman who wants to travel to all 50 states in the U.S. but also wants to minimize the cost of travelling given a starting state. A chromosome presented here contains 50 genes where a single gene is recognized as node or state being visited at a time and the order of genes represents the sequence of states to be visited. A symbol or a number representing the states can be filled into genes depends on how the chromosome will be evaluated. Rules or functions for evaluating each chromosome (solution) are defined accordingly to the requirements and constrains of the problem. In this case, the requirement is minimizing the cost as much as possible. The conformity of a chromosome to the requirement can be evaluated by using rule set or a single function, multiple functions, attest in an actual environment or using the results of a simulator. The mechanism that used to determine the fitness of a chromosome as a solution is called a *Fitness function*.

1.2.2 Reproduction

Reproduction plays an important role for the search techniques because it produces off-springs (new chromosomes or solutions) from existing parents. Off-springs produced in each generation can have as more, less or same quality as a solution as their parents are. This can be visualized as moving from a specific region, represented by previous population, on search space to other specific location, represented by current population. This is one of the processes that move the algorithms toward convergence. Two commonly used genetic operators are *crossover* and *mutation*. These two operators have different role in producing off-springs and will be described in the following section (1.1.2.1 and 1.1.2.2). There are probabilities associated with the operators. These probabilities describe chances for these two operators to take place during evolution.

1.2.2.1 Crossover

Crossover is a genetic operator which uses two parents to produce one or more off-springs. Two parents are selected from a population to exchange their genes and form a complete, new off-spring. By exchanging the genes of parents to create off-spring, the off-springs inherit some properties of their parents. It is not necessarily true that selecting better parents will form better off-spring. It is an assumption that selecting better parents might produce off-spring that inherit some preferred properties and exhibit more conformity to the requirement of the problem. Better parents mean having higher chromosome quality and the population tends to move ascending a peak region. According to the assumption, this would move the population toward the local or global-minima (-maxima) moving the search toward convergence. A problem occurs when some of chromosomes start moving toward local-minima (-maxima). The population may converge on this peak point which sometimes is unacceptable by the termination criteria defined by the problem.

1.2.2.2 Mutation

Mutation is another genetic operator which is commonly used. In this operation, one parent chromosome is selected to produce one or more off-spring. New off-spring is produced by applying some modification to one or more genes in the selected parent. This kind of reproduction would assist in exploring search space by introducing randomness and the ability to not get caught in a local-minima (-maxima). An example of a place that mutation might be potentially useful is when the search starts to converge towards a local-minima (-maxima) and almost all individuals have like content. Since the genes of each individual are alike, crossover might not move then away from the local-minima (-maxima) by randomly altering one gene of a selected individual it might help get the search out of that specific region and jump to others which might be better or worse. In this way, increases the probability not getting stuck in a local minima or maxima and give a more complete search of the search space.

1.2.3 Selection

Selection is one important mechanism for EC. It is the process of selecting individuals in current population to form a new population. Commonly, selection helps to propagate good chromosomes to the next generation while it also preserves diversity in population by selecting other kind of chromosome as well. There is a large number of selection methods used in ECs. Some commonly used selection methods are elitism, roulette-wheel, tournament and generational selection. Sometimes, using only one method of selection may not provide the required number of selected individuals to form a new population. Absent chromosomes can be filled by combining one method of selection with others or generating chromosomes randomly or according to some set of rules. These would help in exploring a larger search space and prevent the population from being concentrated around a local solution when there may be a better, global solution available. Increasing randomness by reducing the number of individuals selected by selection methods also increases diversity in a population increases the ability of the search to discover more solutions from a search space. If the randomness is increased too much, the system may be unlikely to find reasonable solutions. In other words, the system is likely to guess for solutions more than evolving towards a solution. This is a trade-off in forming new populations that designers should take into account.

Elitism selection selects a number of individuals according to a specified number or the percentage of the total number of individuals in the population. Mostly, elitism selection selects only the best solutions available in the population to propagate to the next. This ensures that while the search explores the search space, best solutions found are reserved.

Roulette-wheel selection derives its concept from the roulette game where the entire space of the wheel is divided based on individuals' fitness values. The selection is virtually done by randomly spinning the wheel. The chance of picking an individual is proportional to its fitness value. There are some variations of this kind of selection which perform better in some applications. Figures 1.3 illustrates Roulette-wheel selection.

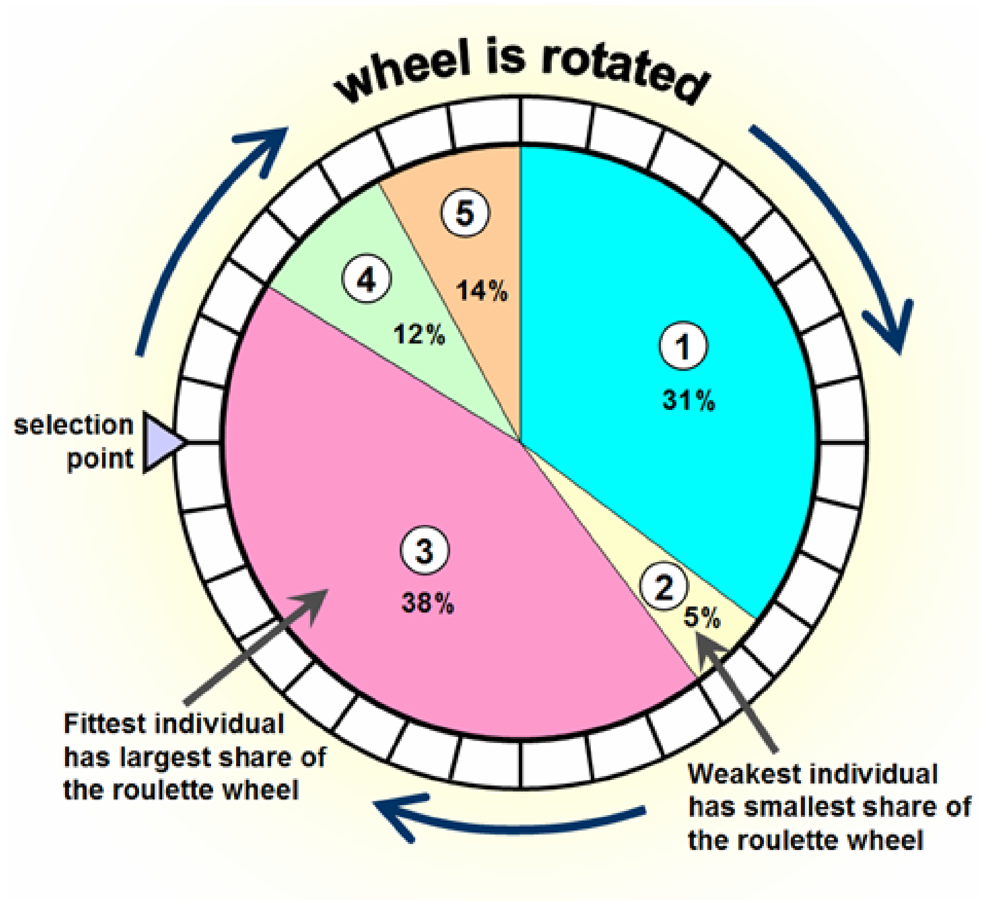

wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Figure 1.3 A Roulette-wheel selection

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

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

1.2.4 Evaluation

Evaluation is a process of measuring quality of a chromosome as a solution to a particular problem. In the evaluation process, fitness values, which can be considered as scores for individuals, will be assigned to all chromosomes in the population. An *Objective Function* is used to define the method for calculating fitness values so that particular chromosome may be ranked against all the other chromosomes. Selection process makes use of fitness value for selecting chromosomes for future generation. Generally, fitness values are mapped onto a real-number scale. Sometimes, the range for fitness values for some applications cannot be determined; only relative magnitudes of the values are take into account to judge individuals in the population. In this case, the fitness values may be turned into percentages of the sum of all fitness values.

1.2.5 Termination Criteria

Termination criteria define how many generations the EC should run before supplying final solutions. Commonly, a search stops when a specified number of generations have been processed (generations based termination criteria) or when the fitness values are high enough for a particular application (fitness based termination criteria). These two termination criteria can also be combined together.

In generations based termination criteria, the search always runs for a certain specified number of generations. The advantage of this type of termination criteria is that we can limit the number of iterations. In some applications, the search process can converge very fast compared to the specified number of generations. Even if an individual that would satisfy the needs of the application, the search still continues the run until it reaches the n-th generation specified in the termination criteria. This results in the wasting of time, power and effort to run the search. Usually, this type of termination criteria is used to find optimal point where search starts to be convergent and we are satisfied with individuals in current population. We can do the tune-up for the number of generations to that optimal point.

In fitness based termination criteria, a minimum value for the fitness value is specified. The search runs until it finds one or more individuals meeting that fitness criteria irrespective of the number of generations that has been processed [27]. If the specified fitness value is too high the search can take a very long time to run even individuals found are applicable for application. If no satisfactory solution is found the system may run indefinitely. We can do a trial-and-error experiment with this type of termination criteria to observe the range of fitness values that will be appropriate for this type of termination criteria.

We can mix the two termination criteria described above to form an intermediate termination criteria. By mixing the termination criteria, the search runs until one or more individuals with fitness values satisfy the minimum fitness value specified, or the number of generation specified is reached. The problems of generations based termination criteria and fitness based termination criteria, which are time wasting and infinite loops respectively, are solved but satisfactory individual is not always guaranteed.

1.2.6 The General flow for algorithm

The processes described in section 1.1.1 – 1.1.5 are the basic processes that are used in general EC applications. Figure 1.4 shows the flow-chart describing the basic flow for EC.

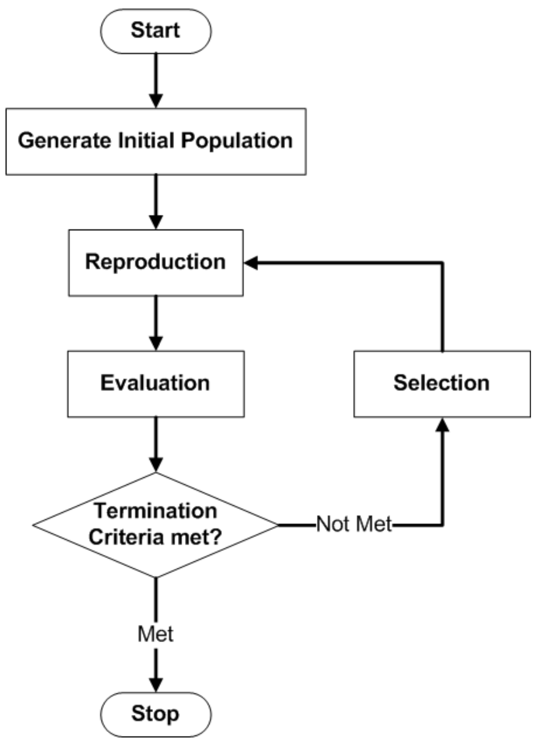


Figure 1.4 The general flow for EC

First, an EC starts by generating initial population. Generating initial population is required and important because it places the search in a specific region on search space which can either be near or far from the optimal solution. Sometimes, placing the search near local-minima (-maxima) can result in the search convergencing quickly to a local-minima (-maxima). Locating the initial population too far from a solution might mean many generations will be needed for convergence. Generating initial population can be done randomly or some knowledge can be applied to the process.

Then next step is reproduction in which crossover and mutation are performed in order to produce off-spring. Both parents and off-spring are evaluated in the evaluation step to obtain fitness values for every chromosome. The termination criteria, previously described in section 1.1.5, are applied to the population after the evaluation to see if an appropriate chromosome has been found. If the termination criteria are not met, the selection process performs individuals' selection to reduce the number of chromosomes that can propagate to the next generation. Without this reproduction the number of chromosomes will be accumulated resulting in processing time and memory problems. After the selection process is completed, the reproduction step is re-visited and crossovers and mutations are performed on the new population. This cycle is repeated until the termination criteria are met.

1.3 Evolutionary Computation on Hardware

1.3.1 Implementation Problems

Evolutionary Computation (EC) has been used to solve many kinds of problems in the real world. The effectiveness of EC consists of its high suitability for parallel computer implementation; particular success in large search and optimization problems and in an ability to learn complex relationships in incomplete datasets [1]. There are many efforts to try to speed-up the process of solution evolution by implementing EC on a reconfigurable hardware platform such as FPGAs [21-24]. An application shown in [1] claims that a hardware-based EC with 1MHz clock-speed operates 2,200 times faster than

EC written in C and running on 100MHz workstation. This can be achieved by implementing the function of parent selection, crossover, mutation and survival in hardware in such a manner that each function can be executed in a single machine cycle.

Implementing an EC system in hardware is a hardship for those without hardware knowledge. To design an EC system knowledge, on low-level hardware interactions and debugging is required. In addition, different EC applications require different EC hardware architectures [24]. This makes changing an EC system in hardware much more difficult when the specifications of the application are changed.

Normally, changing parameters or methodologies used in a software-based EC system such as chromosome size, gene size and population size (or even crossover and mutation methods) requires a good framework and well designed system. Debugging can be done by inserting debugging statements and observe the probed variables through Graphical User Interface (GUI). Changing parameters and methodologies in a hardware-based EC system can be difficult because the user is not only required to have some knowledge on the EC system itself; but is also required to have some knowledge on hardware structure implementation and much forethought is needed to determine debugging channels for a particular hardware structure. Many EC systems have been implemented on a dedicated system designed for a specific application [5-7]. There is also a flexible hardware core for EC system which supports a number of applications [24]. Some of these are implemented on reconfigurable hardware in a Hardware Description Language (HDL). Changing parameters and methodologies can be done relatively easily by altering some parts of modules described by HDL. But this, too, requires knowledge on the particular hardware structure and HDL. A debugger is convenient to have during development of an EC system but the debuggers that are used in HDL development platforms do not offer functionality to easily evaluate characteristics of EC systems. Getting information on the volutionary progress of a system is inconvenient at best with these HDL development tools and the amount of data needed to evaluate. And, even if a debugging system is developed for a particular EC application it will, most likely, not be appropriate for a different EC application.

The purpose of developing the GPEAT system for EC application is to obtain a General Purpose Evolutionary Algorithm Testbed which is flexible and reconfigurable for many EC applications. GPEAT is a substrate for implementing EC applications in hardware. GPEAT also facilitates the implementation of EC applications by providing a Graphical User Interface (GUI) which is used to quickly enter a description of a particular EC system that the user would like to put into hardware. This GUI is called *Parameter entry GUI* which will be described in chapter 3.

Our ultimate goal is to create a substrate for EC application in hardware that requires the user to only have minimal hardware skills to develop EC systems on and is flexible enough that it can be used for any number of applications. The structure for the substrate would be similar to FPGA which has Logic Elements (LE) and Memory Blocks. Some functional elements that could be reconfigured according to user specification would be derived to enable efficient EC implementation. A Graphical User Interface (GUI) for configuring this flexible substrate for EC application would greatly ease the implementation of EC application on hardware and reduce implementation time.

1.3.2 Debugging Problems

Debugging an EC system in hardware is similar to debugging an SoC (System-on-a-Chip). One major problem for debugging SoCs is visibility. Normally, an EC system takes input from users (initial population, parameters for Random Number Generator etc.) and produces outputs which are possible solutions to the problem. If something goes wrong and the system does not produce appropriate outputs, debugging is needed to correct the problem. Since we have no access to components of EC system on silicon, some methodologies such as signal tapping (routing internal nodes to external pins to see what is happening) [28] are needed. To facilitate the debugging process, a debugging platform has to be defined for the GPEAT.

An example of low-cost debugging platform for SoC is proposed in [29] where a dedicated TAM (Test Access Mechanism) controller controls communication between CUDs (Core-Under-Test) and IEEE 1500 modules. "IEEE 1500 standard is a scalable standard architecture for enabling test reuse and integration for embedded cores and associated circuitry" [30]. The CUD in our case is the GPEAT core and the IEEE 1500 modules would be the debugging modules. This is an example which suggests to us how to build a reusable debugging platform for GPEAT system that can be used for any system developed to run on GPEAT.

One may argue that a simulator can also be used for debugging a hardware model before being downloaded to a reconfigurable device. One major problem for using simulator is that the user has to have a great deal of knowledge or experience with the GPEAT system architecture. Users would be required to understand signals integrity in the system. A simulator shows the circuit responses for a fixed time period and, since we cannot determine how long the algorithm would take to converge, it would require a tremendous amount of data logging on a simulator to make sure we have the necessary data for analysis after a run. Users would have a lot of work to get the stored numerical data into a form that would give them useful data.

In this version of GPEAT, a hardware platform for debugging EC systems and debugging software are described. They will assist the user in debugging of the system and to perform performance checks on the implemented EC system. The software only provides simple visualization and data tracking for evolutionary process but makes a complete set of the data from the run available to the user for specific evaluations. One of our goals is to distribute GPEAT core and associated software as an open-source package in research communities. Further improvement can be achieve by distributing the debugging software to groups of users and obtain feedback and additional requirements from users. Different applications may need different requirements and visualizations for observing their evolutionary process, but we believe that some of them share commonality for building a generalized debugging software. Because complete data is made available to the user, building customized debugging software for a particular application is also possible.

1.3.3 Hardware Platforms

The development of EC applications on artificial systems has been investigated for many years. Currently, there are no programmable substrates available made specifically for EC systems though flexible reconfigurable devices are most commonly used as substrates for EC systems in publications. Since most applications for EC are combinatorial optimization problems, (function optimization problems) these problems can be implemented using chromosome representation in binary-string, integer and real-valued form. For example in [24], it is presented that “binary encoding is the most probable form for many of the problem in the industrial engineering world. Real-valued encoding is best used for function optimization problems. Integer encoding is best for combinatorial optimization problem.” These three representations can be implemented on platforms for reconfigurable digital devices and, therefore, we limit our discussion to these platforms. There are three categories of programmable platforms available and they are SPLDs (Simple Programmable Logic Devices), CPLDs(Complex Programmable Logic Devices) and FPGAs (Field Programmable Gate Arrays). The description and characteristic of these three platforms are shown in Table 1.1 to Table 1.3.

SPLD Characteristic	**Range**
Number of pins	16 to 28 pins
Number of macro cells	8 to 24 logic cells
Number of flip-flops (FFs)	8 to 24 FFs
Configuration technology	EPROM, EEPROM
Power-up status	Nonvolatile
Programmability	Can be reprogrammed after being erased
Programming mechanism	Generally programmed off-board
Size	Small

Table 1.1 SPLD Characteristic [25]

CPLD Characteristic	**Range**
Number of pins	44 to 300+ pins
Number of macro cells	32 to 500+ logic cells
Number of flip-flops (FFs)	32 to 500+ FFs
Configuration technology	EEPROM, EPROM, FLASH
Power-up status	Nonvolatile
Programmability	Can be reprogrammed
Programming mechanism	Can be programmed in-circuit
Size	Medium
Equivalent Gate Count	900 to 20,000+ equivalent gates

Table 1.2 CPLD Characteristic [25]

FPGA Characteristic	**Range**
Number of pins	50+
Number of macro cells	5,000+
Number of flip-flops (FFs)	5,000+
Configuration technology	Flash, EEPROM
Power-up status	SRAM: volatile, OTP: nonvolatile
Programmability	SRAM: can be reprogrammed, OTP: no
Programming mechanism	SRAM: can be programmed in-circuit
Size	Medium to Large

Equivalent Gate Count	10,000+ equivalent gates

Table 1.3 FPGA Characteristic [25]

Hardware Description Language (HDL) to describe hardware architecture such as VHDL (VHSIC Hardware Description Language) or Verilog can be used to program these reconfigurable digital platforms. Hardware architecture designed with VHDL or Verilog is synthesized by a Synthesis Tool which compiles and maps HDL codes into an implementation technology for reconfigurable devices of ASIC (Application Specific Integrated-Circuit). Examples for synthesis tools are Xilinx® XST, Synopsis Synplify and Sonata. Then, this synthesized design will be used to generate a configuration bitstream by tools provided by reconfigurable device manufacturers such as Xilinx®, Altera®, Aldec® and Actel®. In this literature, we use a reconfigurable device provided by Xilinx®; Xilinx® XST and Xilinx® ISE for synthesis process and generating configuration bitstream.

The reconfigurable device platform most commonly used in this literature is an FPGA which has a high-density of logic cells. The GPEAT design includes a core which performs general tasks for EC application and an embedded-debugger providing interfaces with host-PC for debugging purpose. The sizes of these two components of GPEAT system vary according to size of the chromosomes and the population size of the system. The component which utilizes the most resources in FPGA is the GPEAT core. As suggested in [26], experiments were conducted on the GPEAT system and the resource usage is shown in Table 1.4. The equivalent numbers of gates used in these particular applications vary from 10,000 to 30,000 gates which suggest that a high-density reconfigurable device must be used.

Table 1.4 Design analysis table [26]

		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 Dual Port RAMs	9,312	1,628 (17%) 1,343 253 32	1,799 (19%) 1,559 218 22	2,212 (23%) 1,927 253 32	1,426 (15%) 1,186 218 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

1.4 Report organization

In Chapter 1, we presented the basic theory for Evolutionary Computation along with the hardware implementation and issues for EC. Chapters 2 present the background for GPEAT and prior works. Specification and functionality of GPEAT is also described. The GUI which is used to configure GPEAT is described in Chapter 3 along with the tool for designers to integrate design of GPEAT into the GUI. Chapter 4 describes the architecture and ASM charts of debugging hardware which is attached to GPEAT core. The Debugging GUI which is the associating software used with debugging hardware is presented in Chapter 5. The results for hardware simulation for debugging hardware and hardware usage in FPGA are reported in Chapter 6. Conclusion and Future work is described in Chapter 7.

CHAPTER 2

REVIEW OF THE EVOLUTIONARY HARDWARE TESTBED (GPEAT)

2.1 Introduction

The GPEAT system is a testbed which implements its hardware framework on a Field-Programmable-Gate-Array (FPGA). GPEAT is a general purpose evolutionary algorithm testbed which allows users to create and run EC systems without hardware knowledge. This claim is supported by analysis of the patterns found in 40 to 50 published papers on EC hardware systems and the information from books and experts. The paper [8-18] were some of the more important papers used in this analysis. The pattern found through the analysis of these references allowed us to find a general framework which all of the systems we examined could be reproduced. This EC flow can be generalized as shown in Figure 2.1.

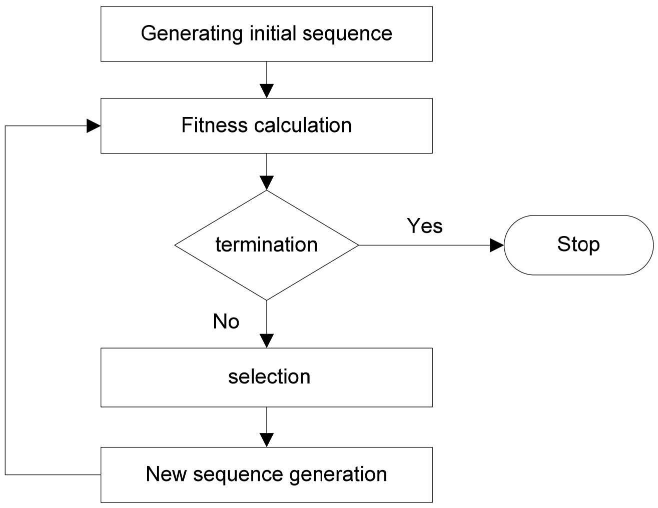


Figure 2.1 Simplified EC hardware flow

Each block of Figure 2.1 [19] has a specified function that was found to be necessary in satisfying at least one of the applications and/or references examined. The initial data is generated by the initial sequence generation block, and is either generated randomly or according to some set of rules. The fitness calculation block supports only intrinsic or extrinsic evaluation on this version of GPEAT but will support internal evaluation according to a user specified rule set by GPEAT in future versions. If the termination criteria is met the system stops. Termination criteria may be a maximum number of cycles through the flow or when the system finds a solution with a suitable fitness. The selection block picks which members of the present iteration through the flow should go on to the next iteration of the flow. The new sequence generation block uses fixed types of genetic operators to create the next set of possible solutions to the problem. To extend this system for flexibility and to give it the ability to be programmed easily to implement a different EC system each and every block shown in Fig 2.1 must be generalized. This generalization is achieved by hardware constructs specific to implementing each EC system block that are reconfigurable through a simple set of parameters and created to specifically carry out the block's job efficiently in time and hardware resources. For example, the new sequence generation block is easily reconfigured to use a variety of genetic operators as specified by the user. The system as a whole must also be flexible. For example, the fitness block should support both intrinsic and extrinsic evaluations. Our system provides all the necessary features needed for reconfiguring to a variety of EC systems.

2.2 GPEAT User Interface and Parameters mapping

The hardware framework is functionally divided into multiple sections. A section will be configured by parameters to be provided by the user. The user needs to be given the ability to enter these parameters by some means which is flexible, user-friendly and no hardware knowledge required. A good user interface is needed for general purpose system in order to hide the details of the hardware and make design and fine tuning of a system easy. These requirements lead to the creation for a Graphical-User-Interface which is called GPEAT user interface (GUI) in this thesis. Our system provides a friendly graphical interface through which the user can communicate easily with the system.

The GUI provides six sets of parameters which are used to configure the hardware framework for a particular EC application. These six groups contain information regarding initial population generation, crossover and mutation control, system control, output control, fitness function information and sensors interface information. Some of these parameters are mandatory while others are optional. This will be discussed in detail in Chapter 3.

As mentioned, the parameters are entered through the GUI. The GUI maps the parameters into a VHDL include file which is accessible by the GPEAT main VHDL code. The Main VHDL, the include file and other essential files are generated by the GUI accordingly to the settings that user has specified. These VHDL sources are then configured using the Xilinx® Design Tools Suit (ISE) for upload to the FPGA. Xilinx® tools are used to partition and place and route design, generate bit files and, finally, downloading designs to the Xilinx® Spartan 3E™ board. This flow is shown in Figure 2.2.

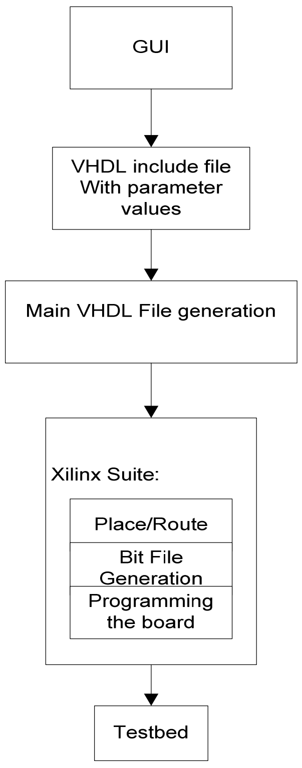
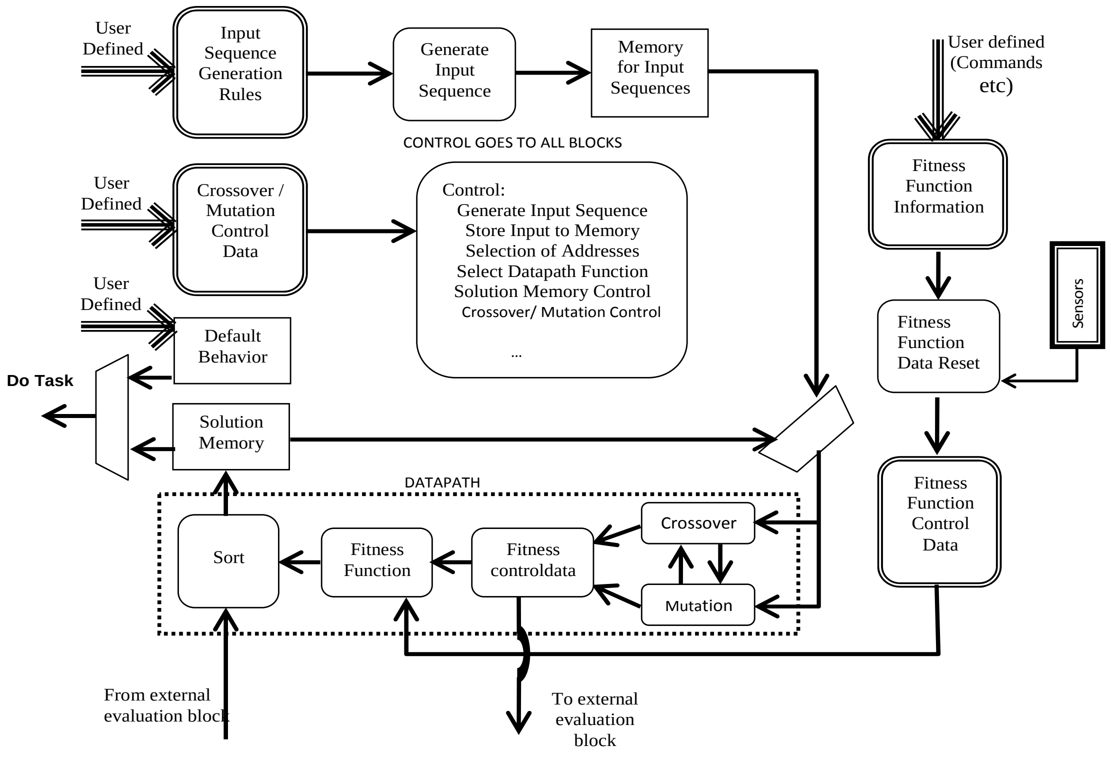


Figure 2.2 GPEAT configuration flow

2.3 Structures for the hardware framework

Based on the EC parameters set through the GPEAT user interface, the FPGA is programmed. The blocks shown in Fig. 2.2 can be divided into the six parameter groups mentioned in the previous section. The initial population generation parameters, crossover and mutation control parameters, central control parameters, output control parameters, fitness function parameters and sensors parameters. The thicker arrows in Figure 2.3 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 instructions from the central control block. The functions of these blocks are explained in the following sections.



User Defined
Input Sequence Generation Rules
Generate Input Sequence
Memory for Input Sequences
User defined (Commands etc)
CONTROL GOES TO ALL BLOCKS
User Defined
Crossover / Mutation Control Data
Control:
Generate Input Sequence
Store Input to Memory
Selection of Addresses
Select Datapath Function
Solution Memory Control
Crossover/ Mutation Control
...
Fitness Function Information
Sensors
User Defined
Default Behavior
Fitness Function Data Reset
Do Task
Solution Memory
DATAPATH
Fitness Function Control Data
Crossover
Sort
Fitness Function
Fitness controldata
Mutation
From external evaluation block
To external evaluation block

Figure 2.3 Hardware Block Diagram

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 (Figure 2.3 [19]). 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 the input chromosome rules generation block's 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. This is one example of where hardware has an advantage over software: As the present population is being evaluated and reproduced, the next, new population members can be generated in parallel. In software the processing would be done in series and one process would not be able to start until previous one was done. The user has the option of setting rules to check the eligibility of the chromosomes generated. In that case, if a check is run, the initial population will be a set of legal chromosomes. If the system does not find a legal chromosomes after a specified number of iterations, the system will show an error message. Memory for the input sequence block stores initial population for that run, as all the chromosomes of initial population are not sent to the next block at once, the remainder must be held until needed. The size of the memory is determined by the size of the population.

2.3.2 Crossover and Mutation Control blocks

In this part of GPEAT, there are three blocks: The crossover/mutation control information database block, crossover/mutation control block and crossover-and-mutation block. The crossover/mutation control information database block stores all information from the GUI regarding the reproduction operations to be performed upon generation of new chromosomes. It also determines the crossover type or mutation type to perform if the user opts for an exclusive reproduction type. The crossover/mutation control block internally controls the two blocks in this group which are crossover and mutation blocks. In the crossover and mutation blocks, user specified crossover and mutations are carried out.

2.3.3 Central Control blocks

The central control database block and central control block are included in this group. Mainly, this group of blocks holds the GPEAT system parameters that have been entered by the user through the GUI and then controls each and every block of the system shown in Figure 2.3. The central control database block specifically holds the GPEAT system parameters which describe the flow of the system and how the central control block should operate the system. The central control block connects to every building block of the GPEAT system. It sends control signals to blocks to activate or disable each block depending on the information stored in the central control data block. The central control block also controls the sequence of operations and data flow between blocks. In the initialization step, the control signal is sent to initial population generation blocks to generate an initial population. After having an initial population, crossover and mutation blocks are activated to perform the reproduction operations. The fitness evaluation blocks are then activated next. Fitness evaluation can be done internally or externally depending on the options that have been chosen by the user. The control signals activate the internal evaluation block if internal evaluation is chosen. Otherwise, the same control signals disables the internal evaluation block and operate the external evaluation part of GPEAT. After finishing the fitness evaluation process, the sorting block and the output block are activated respectively. Again, because this is in hardware, processing is run in parallel as often as possible when there is no conflict.

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 until a solution is found that satisfies the threshold criteria set by the user.

Sorting is the slowest procedure in many systems. The best of the sorting techniques works at O(n log n) [20]. 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. The bins only need to be the size of the number of individuals that might be carried over to the next population (elitism rate x population size). In this way, sorting becomes on the order of O(constant) and the memory needed is only

$$SortMemorySize = 3bins \times (ElitismRate \times Population)\text{s}$$

The best solution by fitness value is always stored separately and will not be lost by overflow of bin 1.

2.3.5 Fitness Function blocks

The fitness function blocks consist of the fitness function information block, fitness function data reset block, fitness function control data block and fitness function calculation block. These blocks are used to determine fitness value for chromosome based on specified parameters by user. Fitness values calculation can either be done internally or externally as explained in the previous sections (section 2.3.3.). In an internal evaluation, a set of rules defined by the user is stored in the fitness function information block. These rules are used by the fitness function data reset block and fitness function control data block to realize an objective function. Information from sensors can be used as input to objective function to make a dynamic objective function which evaluates chromosomes according the present operating environment. The fitness function block performs calculation described by the realized objective function. In an external evaluation, chromosomes stored in GPEAT system are sent out to some external evaluating environment. These external components then return the fitness values of chromosomes back to GPEAT system in system-usable format. When a chromosome is being sent out from the GPEAT system, it is sent out along with a start signal to the external evaluating component. The start signal synchronizes chromosome switching with chromosome acquisition process of the external component. When fitness value is completely calculated by the external component, the external component will send the fitness value back to the GPEAT system along with a done signal so that GPEAT knows that a fitness value has been found for the chromosome under test and that fitness value should be stored.

2.3.6 Sensors blocks

The use of sensor block is not available in this version of GPEAT. The GPEAT system breaks the barrier between the EC hardware system and real-life events by providing a bread board area to connect sensors to provide information about the environment. Sensors which are connected through the sensor block collect the raw data from operating environment and that information can be used for fitness values calculation. The objective function created for the fitness values calculation can be set up to change accordingly to the operating environment to find appropriate chromosomes at a particular time. Data from the sensors need to be converted into a system-usable format by some means such as using an analog-to-digital converter (ADC), a decoding block or some type of amplification. This conversion is needed because of the wide variety of sensors. For example, if a thermal sensor was used for a particular application, the output from those sensors would be analog and so must be converted to binary through an analog to digital converter. Once converted to a binary form the data can be used for processing.

2.4 Debugging GPEAT system

When a GPEAT system is configured for a particular application, the user provides configuration information to the system. Sometimes the user might never have envisioned the problems that would come with their settings. For an application, the configuration and development of the system may involve a trial-and-error process which may require many iterations of setting parameters and then watching the results. Besides providing inputs and obtaining outputs from the system, the user might want to see the intermediate states and how the internal mechanisms are working in order to explain why a particular output resulted, or to tune up the system for optimal performance.

GPEAT configuration flow in Figure 2.2 but, in addition to the basic EC system, GPEAT also provides a debugging interface for the user to obtain information and trends of the system during runs. GPEAT has a built-in debugging hardware interface used for obtaining and verifying the system parameter information and all population- and chromosome-data. The debugging hardware sends information back to host PC where a debugging Graphical-User-Interface runs. Information is sent back from the system through an RS-232 connection and where it is displayed in graphical form. Figure 2.4 shows the connection diagram for GPEAT debugging interface.

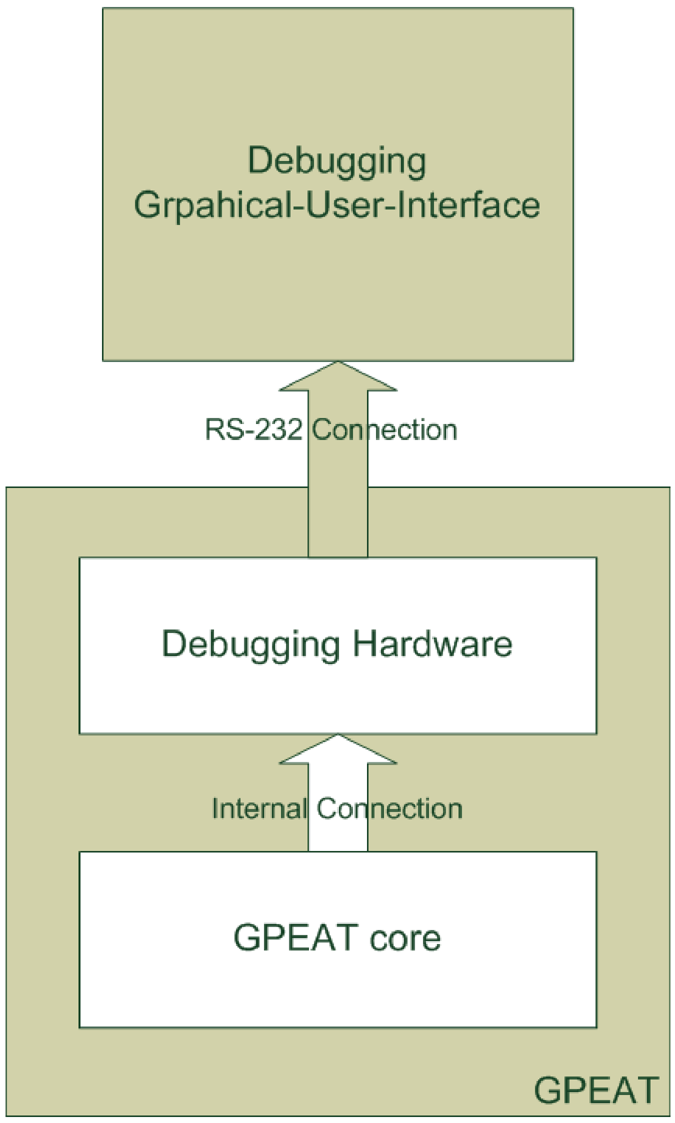


Figure 2.4 Debugging connection diagram

CHAPTER 3

GRAPHICAL USER INTERFACE DESIGN

3.1 Parameters Entry GUI

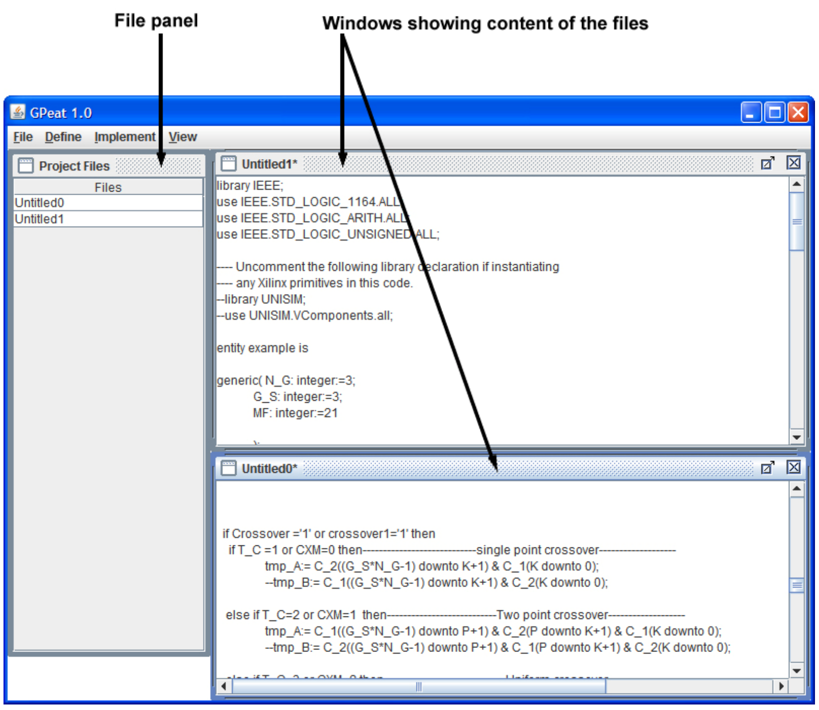
The Parameters Entry GUI is used to configure the GPEAT system. It provides a graphical interface for user to enter the parameters to configure GPEAT system for a particular application. Some of these parameters are required and some are optional but they all are used to generate VHDL codes which are used to construct the GPEAT core and built-in debugging hardware. As shown in Figure 2.2, the GUI generates GPEAT core accordingly to the parameters that a user has specified by mapping the parameters into a VHDL include file. Then, these VHDL files are joined in an FPGA project file using Xilinx® Design Tools Suit (ISE) to generate a FPGA configuration file (.bit file).

The GUI stores the information for the VHDL codes and parameter mapping into a single binary file (GPEAT.dat). When the GUI generates the VHDL codes for the GPEAT core and debugging hardware, it reads this binary file to obtain information regarding the codes generation such as the amount of file required to construct GPEAT, the content of the codes and information for parameter mapping. The binary file can be generated by a GPEAT developer using *GPEAT binary data-file encoder* which is another GUI used together with the parameter entry GUI. The binary data-file encoder will be described in section 3.2.

The parameter entry GUI separates parameters into five categories which are Input (Sensors), Default Output, Initial Population, Fitness Rules and Debugger. Each of these categories is described in following section (section 3.1.2-3.1.7). First, the workspace of the GUI will be described in section 3.1.1.

3.1.1 Workspace

The workspace shows the files which contain VHDL codes for the GPEAT core and debugging hardware. It contains a file panel, which summarizes the files required for constructing system and separate windows showing content of the files. These windows will be visible only when the user chooses to view the contents of the files. If user wants to view the contents of a file but the window showing the contents of the file does not appear, the user can recall the window by doing double-click on the file listed in file panel and then the window of the file will appear. Figure 3.1 shows workspace of the GUI.



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

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity example is

generic( N_G: integer:=3;
         G_S: integer:=3;
         MF: integer:=21
```

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

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

Figure 3.1 Workspace

3.1.1.1 Creating/Loading/Saving Project

Before working with the parameter entry GUI, the user has to create or load a project. A project file is named with the extension ".gpt" and stores values for the GPEAT parameters. Creating a new project initializes all GPEAT parameters to default values and then the user has to specify the parameter values they want to override those default values manually. This will be discussed in section 3.1.2 to 3.1.7. Saving a project will allow the user to continue work later or save the values of the parameters in a file. The user can create, load and save project by clicking *Files→Project→New Project, Files→Project→Open Project* and *Files→Project→Save Project* respectively.

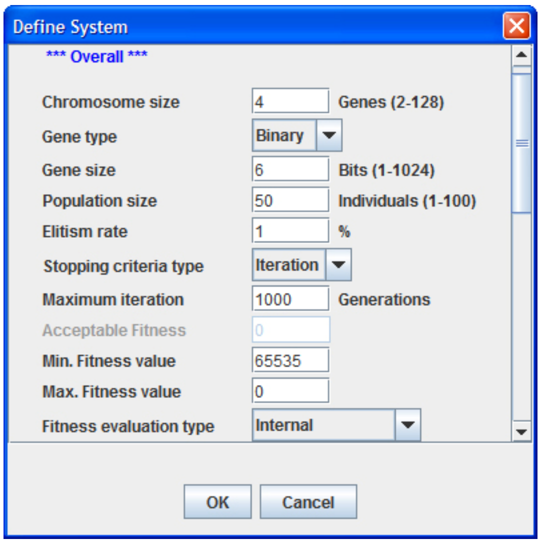
3.1.1.2 Creating/Loading/Saving a Source

User can also add additional sources as required in addition to the GPEAT core codes. The added code can be used to work with GPEAT as, for example, an intrinsic testing block would need to do. The GUI does not help the user create these additional blocks and the user would have to have hardware design skills in order to create them. The user can create, load and save added source by clicking *Files → New Source, Files → Open Source* and *Files → Save Source/Save Source as* respectively.

3.1.2 System

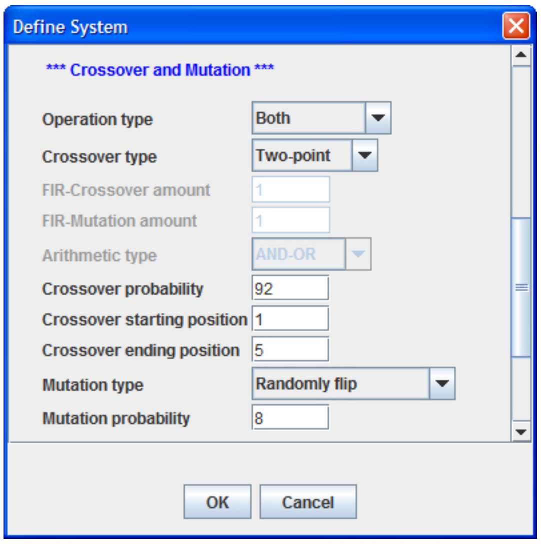
The system parameters window can be access by clicking *Define → System* from the menu bar of the GUI. The system parameters are used to define the input/output of the system, operations during runs and the behavior of the GPEAT system. Some of these parameters are essential for VHDL generation because they are used to define the structure of GPEAT system for a particular application. In the

Define System window, the window is divided into three sections grouped by the functions of the parameters. Figure 3.2(a), Figure 3.2(b) and Figure3.2(c) show three sections of Define System window which are the overall section, the crossover and mutation section and the random number generator section.



*** Overall ***

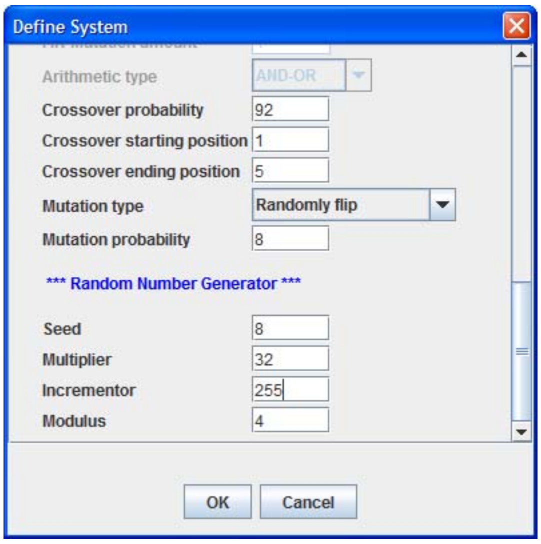
Chromosome size	4	Genes (2-128)
Gene type	Binary	
Gene size	6	Bits (1-1024)
Population size	50	Individuals (1-100)
Elitism rate	1	%
Stopping criteria type	Iteration	
Maximum iteration	1000	Generations
Acceptable Fitness	0	
Min. Fitness value	65535	
Max. Fitness value	0	
Fitness evaluation type	Internal	



*** Crossover and Mutation ***

Operation type	Both
Crossover type	Two-point
FIR-Crossover amount	1
FIR-Mutation amount	1
Arithmetic type	AND-OR
Crossover probability	92
Crossover starting position	1
Crossover ending position	5
Mutation type	Randomly flip
Mutation probability	8

(a) Overall section (b) Crossover and mutation section



Arithmetic type	AND-OR
Crossover probability	92
Crossover starting position	1
Crossover ending position	5
Mutation type	Randomly flip
Mutation probability	8

*** Random Number Generator ***

Seed	8
Multiplier	32
Incrementor	255
Modulus	4

(c) Random number generator section

Figure 3.2 System parameters window

In the overall section, the parameters which define the structure of the GPEAT system are set. For example: the chromosome size, gene size, gene type and population size are grouped in this section. These parameters are the same parameters shown in Figure 3.2(a) and are described and summarized below in Table 3.1 and Table 3.2 respectively.

Parameter	**Description**
Chromosome size	The length of chromosome (number of genes contained in a chromosome).
Gene type	Specify data type contained in each gene. This can be a binary number or integer number (32-bit).
Gene size	Specify number of bits contained in a gene for binary typed gene.
Population size	Specify number of chromosomes in a population (each iteration)
Elitism rate	Specify number of best chromosomes in percent that will propagate to next generations.
Stopping criteria type	Specify condition for system to stop run. It can based on number of iteration or fitness-value based.
Maximum iteration	The maximum number of generations processed.
Acceptable Fitness	The acceptable fitness value (for termination), if Fitness-based termination criteria are chosen.
Min. Fitness Value	Minimum Fitness Value that will possibly be found in the

	system. Maps to 0% fitness value
Max. Fitness Value	Maximum Fitness Value that will possibly be found in the system. Maps to 100% fitness value.
Fitness Evaluation type	Selects module for fitness evaluation. The type can be internal module or external module.

Table 3.1 System Parameter descriptions

Parameter	**Values**	**Default Value**	**Unit**
Chromosome size	2 to 128	8	Genes
Gene type	Binary / Integer	Binary	
Gene size	1 to 1024	1	Bits
Population size	1 to 100	20	Chromosomes
Elitism rate	1 to 100	10	Percent
Stopping criteria type	Iteration / Fitness	Iteration	
Maximum iteration	User specified	0	
Acceptable Fitness	User specified	0	
Min. Fitness Value	User specified	0	
Max. Fitness Value	User specified	0	
Fitness Evaluation type	Internal / Intrinsic-Extrinsic	Internal	

Table 3.2 System Parameters

3.1.3 Input (Sensors)

This category of parameter has not been implemented in the GUI because the sensors blocks (previously explained in chapter 2) have not been completed but will be completed for the next version of GPEAT. This category of parameters is for configuring sensors inputs to the GPEAT system and is used as a part for fitness evaluation process. Users can add sensors to the system through the GUI. The user must specify the type of sensors and its type of input. The parameters for each sensor are summarized in Table 3.4.

Parameter	**Description**
Sensor Name	The name for sensor which will be used as signal name in VHDL code.
Type	Output type of sensor (analog or digital).
Number of Bits (Digital)	Number of bits from output of sensor.
Supply Voltage	Supply voltage to the sensor.
Upper Limit for sensor (Analog)	Maximum voltage or current from output of analog sensor.
Lower Limit for sensor (Analog)	Minimum voltage or current from output of analog sensor.
FPGA I/O Name	FPGA Input/Output name used for Xilinx® Design Tool Suit.

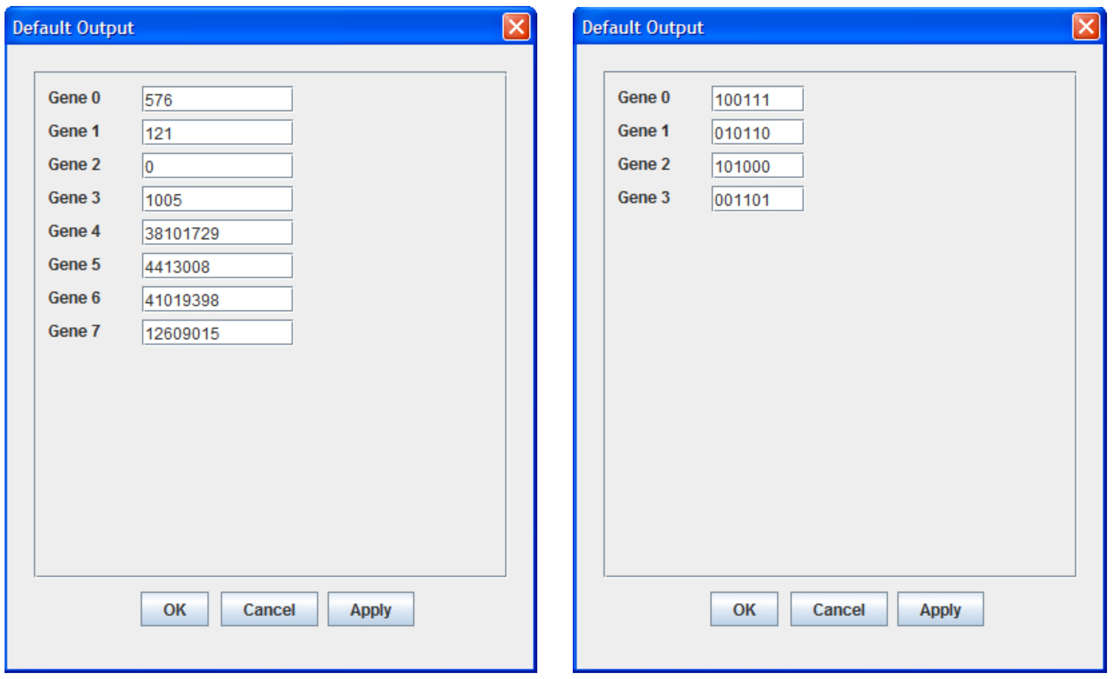
Table 3.3 Input parameters description

Parameter	**Values**
Sensor Name	User specified
Type	Digital/Analog
Number of Bits (Digital)	4/8/10/12/16
Supply Voltage	3.3V/5V/10V/12V
Upper Limit for sensor (Analog)	User specified
Lower Limit for sensor (Analog)	User specified
FPGA I/O Name	User specified (depends on the model of FPGA)

Table 3.4 Parameters for sensor

3.1.4 Default Output

This category of parameters is used for defining a default output of system in case there is no chromosome satisfies the “acceptable solution” criteria of system. The size of default output depends on the parameters in the first category (system). The user must specify the size of the genes and chromosomes and also the type of gene which will result in a specific size and type of default output chromosome. Figure 3.3(a) and Figure 3.3(b) show two example windows for two separate default output specifications. Figure 3.3a shows the settings of a system where the gene size = 32, the chromosome size = 8 and the gene type = integer. Figure3.3b shows the settings of a system where the the gene size = 6, the chromosome size = 4 and the gene type = binary.



Gene 0	576
Gene 1	121
Gene 2	0
Gene 3	1005
Gene 4	38101729
Gene 5	4413008
Gene 6	41019398
Gene 7	12609015

Gene 0	100111
Gene 1	010110
Gene 2	101000
Gene 3	001101

(a) Integer gene (b) Binary gene

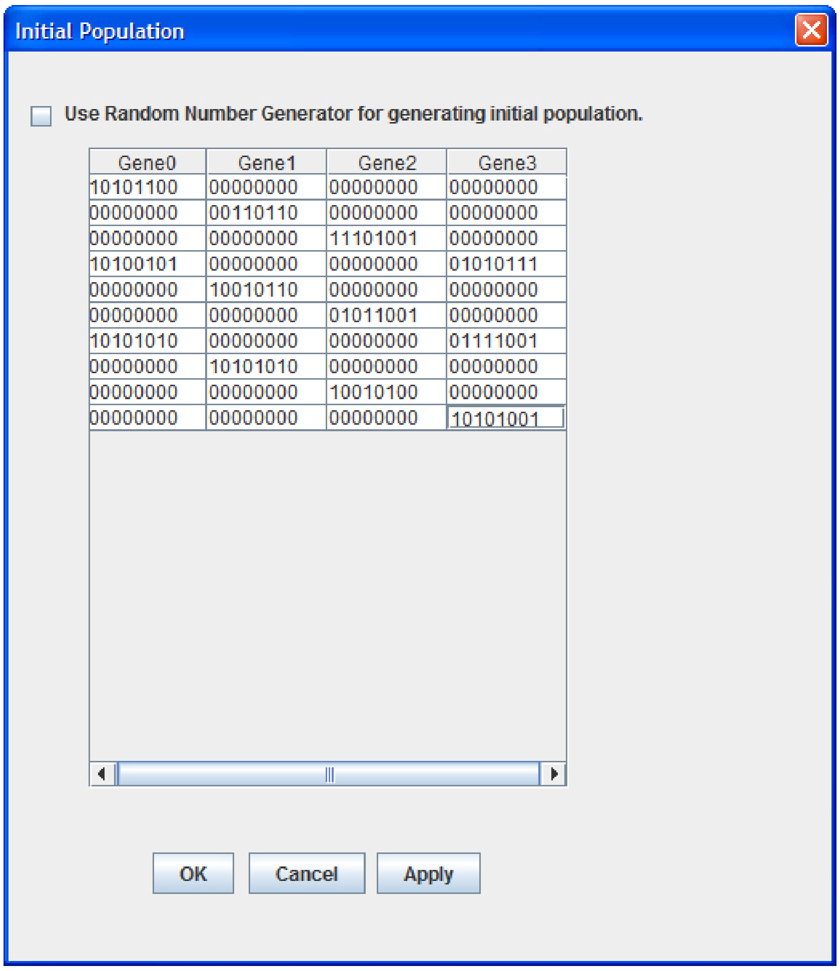
Figure 3.3 Default output examples for two types of gene

As shown in Figure 3.3(a) and Figure 3.3(b), the number of boxes that the user can enter integer or binary values into depends on the chromosome size (number of genes) set in the system parameters category. The number of digits that the user can specify in each box also depends on gene size specified in the system parameters category for binary gene. For an integer gene, the maximum number of digits that can be entered is nine digits.

For the case that user does not specify numbers in every box, the empty box will be automatically filled with a zero value for integer genes and an all-zero bit-string for binary gene.

3.1.5 Initial Population

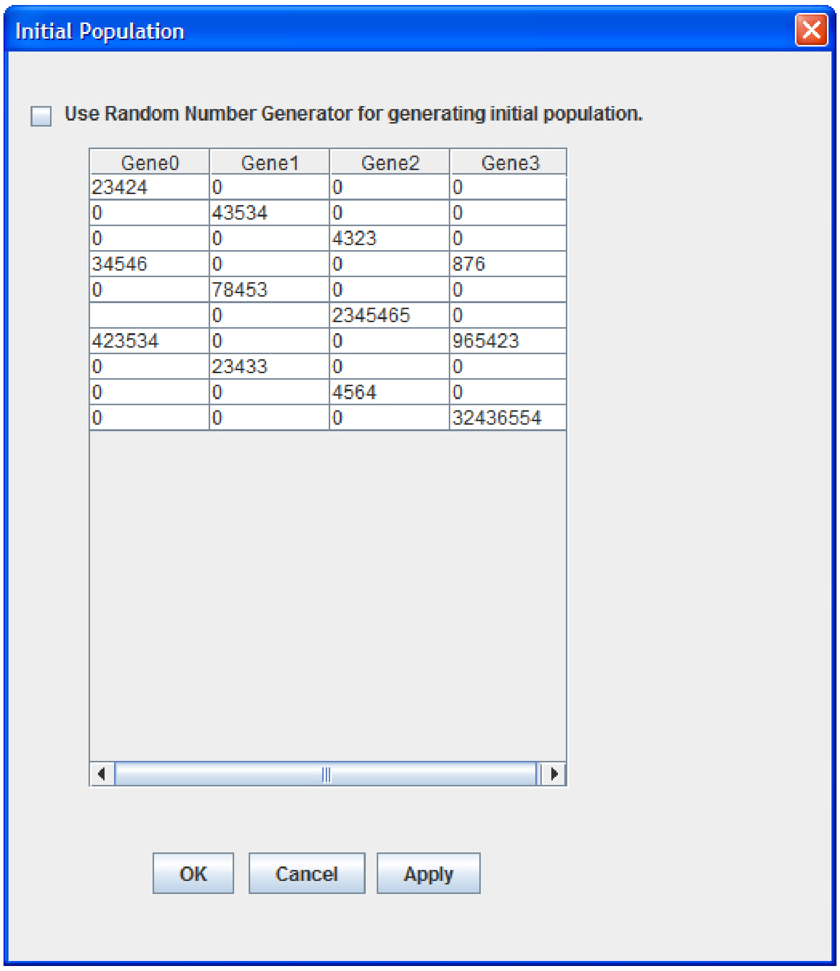
The Initial Population window can be accessed by clicking *Define* → *Initial Population*. This window is used for defining the first population that will be generated in a GPEAT system. If the user does not specify any chromosomes in this window the random number generator will be used for generating chromosomes in the initial population. Otherwise, a set of chromosomes specified in this window will be used for initial population. The window for defining initial population is shown in Figure 3.4 and Figure 3.5.



☐ Use Random Number Generator for generating initial population.

Gene0	Gene1	Gene2	Gene3
10101100	00000000	00000000	00000000
00000000	00110110	00000000	00000000
00000000	00000000	11101001	00000000
10100101	00000000	00000000	01010111
00000000	10010110	00000000	00000000
00000000	00000000	01011001	00000000
10101010	00000000	00000000	01111001
00000000	10101010	00000000	00000000
00000000	00000000	10010100	00000000
00000000	00000000	00000000	10101001

Figure 3.4 An example of initial population for binary type of gene


Initial Population

☐ Use Random Number Generator for generating initial population.

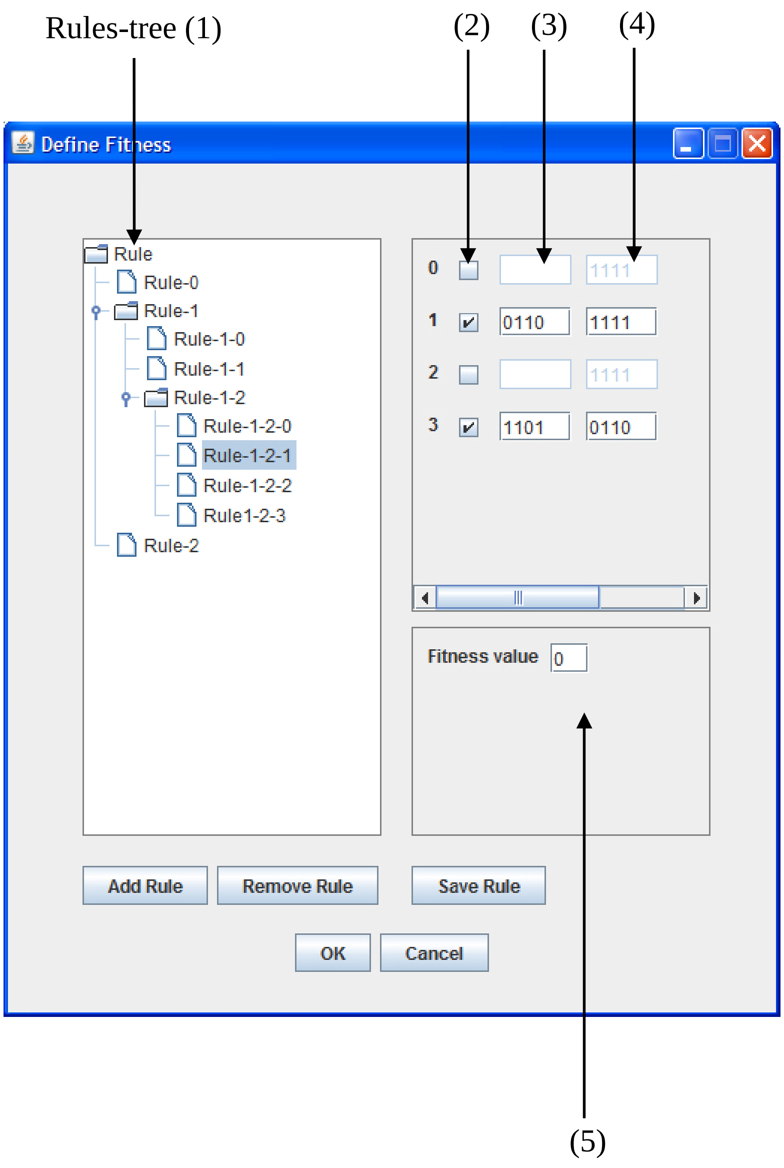
Gene0	Gene1	Gene2	Gene3
23424	0	0	0
0	43534	0	0
0	0	4323	0
34546	0	0	876
0	78453	0	0
	0	2345465	0
423534	0	0	965423
0	23433	0	0
0	0	4564	0
0	0	0	32436554

Figure 3.5 An example of initial population for integer type of gene

An example of binary type of gene (chromosome size = 4, gene size = 8, population size = 10) and an example of integer type of gene (Chromosome size = 4, Population = 10) are shown in Figure 3.4 and Figure 3.5 respectively. The number of genes (chromosome size) specifies the number of columns in the frame and number of rows is specified by the size of population. Thus, chromosomes in this window are viewd by looking along a row. The checkbox above the frame for defining chromosomes is used if user wants to ignore the initial population entered and use random number generator instead.

3.1.6 Fitness Rules

Fitness rules are used for defining rules on how a chromosomes in a population will be evaluated if internal evaluation is selected. This depends on the application that will be run on GPEAT. The fitness rule window can be accessed by clicking *Define* → *Fitness Rules*. This window provides the user the ability to define a set of rules to evaluate the fitness of chromosomes. Figure 3.6 shows the Fitness Rules window.

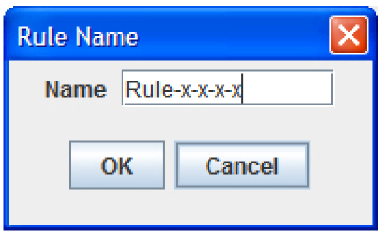


0	☐		1111
1	✓	0110	1111
2	☐		1111
3	✓	1101	0110

Fitness value 0

Figure 3.6 The Fitness Rules window

The rules for chromosomes evaluation are implemented as a rules-tree which is similar to IF-THEN-ELSE statements in programming languages. In figure 3.6, (1) shows implemented rules in a rules-tree. A rule can be defined as an ancestor or descendant of another rule or it can be both. For example, according to the figure, *Rule-1* is the ancestor of, for example, *Rule-1-0, Rule-1-1, Rule-1-2, Rule-1-2-0. Rule-1-2-1, Rule-1-2-2 and Rule-1-2-3. Rule-1-2-0, Rule-1-2-1. Rule-1-2-2 and Rule-1-2-3* are the descendants of *Rule-1-2*. All rules defined by the user has at least one ancestor in common and that is the root-rule and named *Rule* in Figure 3.6 at the top of the window. A rule is defined by selecting an ancestor and then clicking on *Add Rule*. A window for specifying the name of the rule will appear as shown in Figure 3.7. Once a rule is defined then chromosomes matching the pattern specified in the fules will be evaluated according to the If-Then clause statements. For example, a chromosome will be evaluated by a set of rules which has a root-rule named *Rule*. First the chromosome will be examined using *Rule-0*, *Rule-1* and *Rule-2* in that order. If the chromosome falls into *Rule-1,* the chromosome will further be examined with *Rule-1-0*, *Rule-1-2* and *Rule-1-2*. Then, if after being evaluated using Ruls-1, the chromosome also falls into Ruls-1-2, then it will be examined using the set of descendant rules of *Rule-1-2* which are *Rule-1-2-0, Rule-1-2-1, Rule-1-2-2* and *Rule-1-2-3*. The descendant rules of *Rule-1-2* are the leaf-node rules in which a fitness value will be assigned to the chromosome. Final fitness values will be assigned to a chromosome at the leaf-node rule.



Rule-x-x-x-x

Figure 3.7 The window for specifying name for a rule

In Figure 3.6, the user selects genes that are to be examined by a particular rule by checking the checkboxes in the column of (2). The user can specify the contents of genes that will be categorized into a rule in (3). (4) is the gene mask and used to indicate which bit in a gene will be considered in an examination. (5) is for specifying fitness value for chromosome which falls into a particular rule. This fitness value will be assigned to a chromosome when the rule is a leaf-node rule and matches the chromosomes. The default fitness value for a chromosome that does not fall into a rule is zero.

3.1.7 Debugger

This category of parameters indicates whether the debugging hardware should be enabled or not. It also notifies the GPEAT core whether to run continuously or in the single-step mode. The window for this category of parameters can be accessed by clicking *Define* → *Debugger*. Figure 3.8 shows the window of the parameters for debugger.

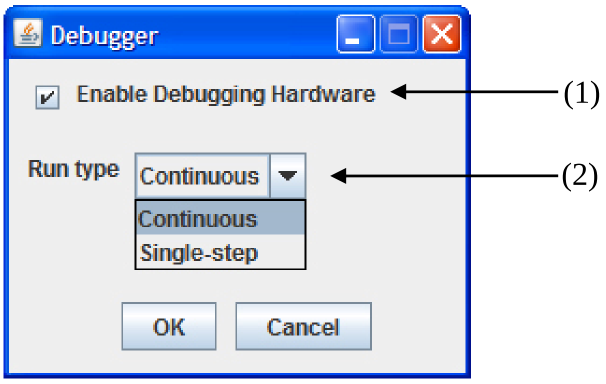
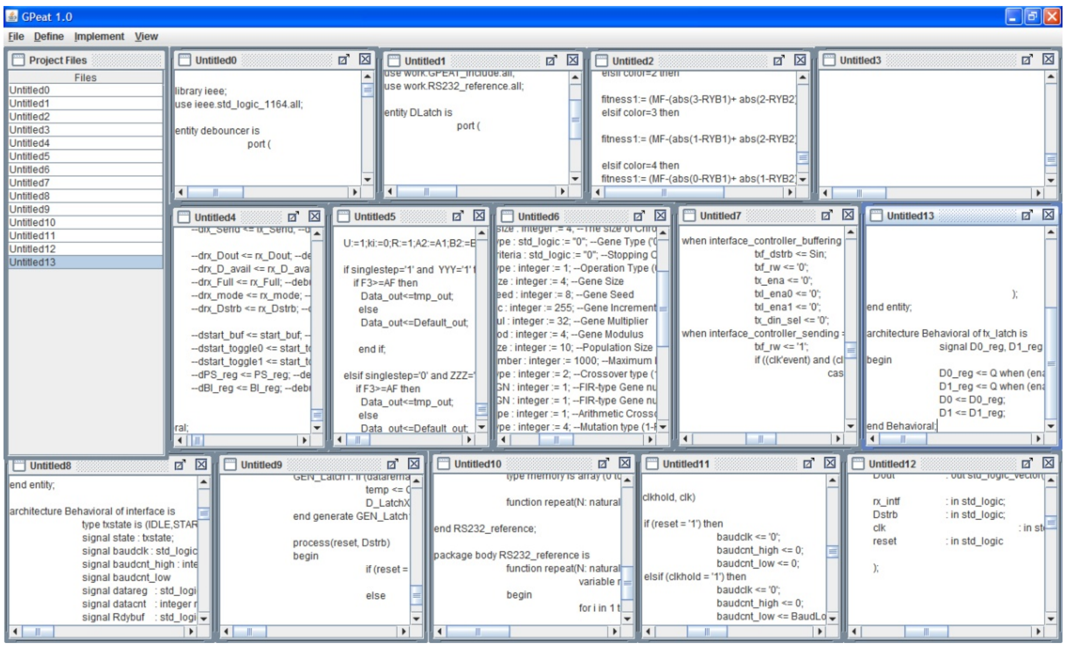


Figure 3.8 Debugger window

The debugging hardware can be enabled by checking the check box (1). This will indicate that the debugging hardware is to be included in the design and should send data back to the computer. The type of run can be selected from the drop-down list (2) to indicate to the GPEAT system whether to run continuously or in the single-step mode. If a single-step run is selected, the user must press a button or trigger FPGA I/O in order to continue to the next step.

3.1.8 Generating GPEAT core

After the user is finished setting all the parameters for a particular application, the next step is to generate the VHDL code required to realize the GPEAT system for the application. A user can use the parameter entry GUI to generate the VHDL code by clicking *Implement* → *Generate*. Then the codes will be generated and all parameters set will be mapped into the code as shown in Figure 3.9.



Untitled0
```
library ieee;
use ieee.std_logic_1164.all;

entity debouncer is
                port (
```

Untitled1
```
use work.RS232_reference.all;

entity DLatch is
                port (
```

Untitled2
```
fitness1:= (MF-(abs(3-RYB1)+ abs(2-RYB2)
elsif color=3 then

fitness1:= (MF-(abs(1-RYB1)+ abs(2-RYB2)

elsif color=4 then
fitness1:= (MF-(abs(0-RYB1)+ abs(1-RYB2)
```

Untitled3

Untitled4
```
    --drx_Dout <= rx_Dout; --de
    --drx_D_avail <= rx_D_ava
    --drx_Full <= rx_Full; --deb
    --drx_mode <= rx_mode; --
    --drx_Dstrb <= rx_Dstrb; --

    --dstart_buf <= start_buf; --
    --dstart_toggle0 <= start_t
    --dstart_toggle1 <= start_t
    --dPS_reg <= PS_reg; --de
    --dBI_reg <= BI_reg; --deb


ral;
```

Untitled5
```
  U:=1;ki:=0;R:=1;A2:=A1;B2:=B

  if singlestep='1' and  YYY='1' t
    if F3>=AF then
      Data_out<=tmp_out;
     else
      Data_out<=Default_out;

     end if;

  elsif singlestep='0' and ZZZ='
    if F3>=AF then
      Data_out<=tmp_out;
     else
      Data_out<=Default_out;
```

Untitled6
```
ype : std_logic := "0"; --Gene Type ('
riteria : std_logic := "0"; --Stopping C
ype : integer := 1; --Operation Type (
ze : integer := 4; --Gene Size
eed : integer := 8; --Gene Seed
c : integer := 255; --Gene Increment
ul : integer := 32; --Gene Multiplier
od : integer := 4; --Gene Modulus
ze : integer := 10; --Population Size
mber : integer := 1000; --Maximum
ype : integer := 2; --Crossover type (
GN : integer := 1; --FIR-type Gene nu
GN : integer := 1; --FIR-type Gene nu
pe : integer := 1; --Arithmetic Cross
ype : integer := 4; --Mutation type (1-
```

Untitled7
```
when interface_controller_buffering
                txf_dstrb <= Sin;
                txf_rw <= '0';
                tx_ena <= '0';
                txl_ena0 <= '0';
                txl_ena1 <= '0';
                tx_din_sel <= '0';
when interface_controller_sending =
                txf_rw <= '1';
                if ((clk'event) and (cl
                                    cas
```

Untitled13
```
                                        );
end entity;

architecture Behavioral of tx_latch is
                signal D0_reg, D1_reg
begin
                D0_reg <= Q when (en
                D1_reg <= Q when (en
                D0 <= D0_reg;
                D1 <= D1_reg;
end Behavioral;
```

Untitled8
```
end entity;

architecture Behavioral of interface is
                type txstate is (IDLE,STAR
                signal state : txstate;
                signal baudclk : std_logic
                signal baudcnt_high : inte
                signal baudcnt_low
                signal datareg   : std_logi
                signal datacnt   : integer r
                signal Rdybuf    : std_logi
```

Untitled9
```
                        temp <= 
                        D_LatchX
        end generate GEN_Latch

        process(reset, Dstrb)
        begin
                        if (reset =

                        else
```

Untitled10
```
        function repeat(N: natural

end RS232_reference;

package body RS232_reference is
        function repeat(N: natural
                        variable r
        begin
                        for i in 1 t
```

Untitled11
```
clkhold, clk)

if (reset = '1') then
                baudclk <= '0';
                baudcnt_high <= 0;
                baudcnt_low <= 0;
elsif (clkhold = '1') then
                baudclk <= '0';
                baudcnt_high <= 0;
                baudcnt_low <= BaudLo
```

Untitled12
```
    rx_intf          : in std_logic;
    Dstrb            : in std_logic;
    clk                              : in st
    reset            : in std_logic

    );
```

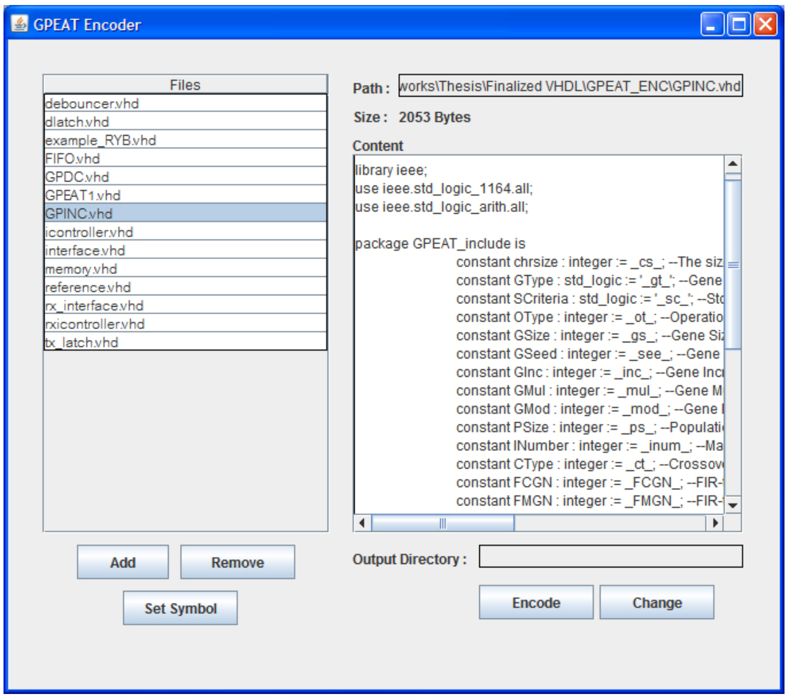
Figure 3.9 Workspace containing VHDL codes generated

The user is required to save the generated VHDL code in some working directory which can be accessed by Xilinx® Design Tool Suit (ISE). ISE is used to synthesize VHDL code and generate the FPGA configuration file.

3.2 Binary Data File Encoder (.dat Encoder)

As discussed in section 3.1, the parameters entry GUI loads all information related to the code generation and parameters for mapping from a binary file (GPEAT.dat). This file contains two parts which are the header and body. The header part stores information about the size of VHDL code and the offsets of the code stored in the body part of the file. The header also stores information on parameter mapping. The body part stores the VHDL code. All VHDL files created by the designer of the GPEAT system will be stored together in this part. The GPEAT.dat file packs all the VHDL files in itself.

The GPEAT binary file encoder is used by the designer to store all required files for realizing GPEAT system into the GPEAT.dat file. It generates a binary file which is understandable by the parameter entry GUI so it can be saved and reloaded when needed. The GPEAT binary file encoder is shown in Figure 3.10.



Files:
debouncer.vhd
dlatch.vhd
example_RYB.vhd
FIFO.vhd
GPDC.vhd
GPEAT1.vhd
GPINC.vhd
icontroller.vhd
interface.vhd
memory.vhd
reference.vhd
rx_interface.vhd
rxicontroller.vhd
tx_latch.vhd

Path : works\Thesis\Finalized VHDL\GPEAT_ENC\GPINC.vhd

Size : 2053 Bytes

Content

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

package GPEAT_include is
                constant chrsize : integer := _cs_; --The siz
                constant GType : std_logic := '_gt_'; --Gene
                constant SCriteria : std_logic := '_sc_'; --Sto
                constant OType : integer := _ot_; --Operatio
                constant GSize : integer := _gs_; --Gene Si
                constant GSeed : integer := _see_; --Gene
                constant GInc : integer := _inc_; --Gene Inc
                constant GMul : integer := _mul_; --Gene M
                constant GMod : integer := _mod_; --Gene
                constant PSize : integer := _ps_; --Populati
                constant INumber : integer := _inum_; --Ma
                constant CType : integer := _ct_; --Crossov
                constant FCGN : integer := _FCGN_; --FIR-
                constant FMGN : integer := _FMGN_; --FIR-
```

Output Directory :

Figure 3.10 GPEAT binary file encoder

As shown above in Figure 3.10, the designer gathers all the design files into the encoder by clicking *Add* or removes some files from the encoder by clicking *Remove*. On the right column of the encoder shows information of each individual file and its content. The designer starts the encoding process by clicking *Encode.* When *Encode* is pushed, all required files for GPEAT system are packed together and the binary file (GPEAT.dat) is generated.

CHAPTER 4

DEBUGGING HARDWARE FOR EVOLUTIONARY HARDWARE TESTBED

4.1 Introduction

In this chapter, we will introduce the concept of debugging a GPEAT system. To make the GPEAT debugging system, a graphical user interface was created that displays progress of the system towards a solution and gives the user the ability to view all data generated by the system including chromosome values and fitness values. To gather the data on the hardware side, a debugging (hardware) core was created that interacts with the GPEAT core and gathers and formats the data for transmission to the computer. To transfer the data between the hardware side to and from the computer and interface based on the UART protocol was also implemented. The debugging hardware interacts with host PC through RS-232 connection which is standardized, easy-to-setup and has moderate data transfer speed. The architecture of debugging and transmission hardware is described by six distinct modules which are integrated to serve as a data-transfer controller between GPEAT system and host PC. The data-transfer format for the debugging hardware and host PC is also described in this chapter.

4.2 Essentials for Debugging Interfaces

The principle for debugging software is to insert debugging statements into source code to observe values of variables to gauge the progress of complicated tasks. In hardware debugging, a group of hardware modules (debugging hardware) responsible for debugging is inserted into the design to do the observation. Two examples of where this type of approach is used is the insertion of test access mechanism (TAM) and wrappers into SoCs core [29, 30].

The main functionality of debugging hardware is to provide designers visibility inside hardware core with minimal disturbance to the core. Some information must be supplied or pulled out from hardware core through debugging hardware; so designers can analyze responses of the hardware core. While providing and supplying information to the hardware core, debugging hardware should not alter the functionality of the core. If the debugging hardware did alter the core the designer would not be able to determine the functional verification of hardware core precisely. Another feature that is required for most applications is that the debugging hardware should consume a minimal amount of resources. If designers desire to attach debugging hardware onto core, the debugging hardware should reduce the availability of resources that could be provided to the core as little as possible.

4.3 The RS-232 Connection

The RS-232 serial communication protocol is a standard protocol used in asynchronous communication. The protocol was defined by a standards committee known as the Electronic Industries Association.

The communication lines used here are called TX-line (Transmitting line) and RX-line (Receiving line). Note that, due to duplex data transmission, the TX-line may be considered as the RX-line in from another peripheral's perspective. It all depends on what peripheral that we use as a reference. In this thesis, we consider the debugger hardware as the reference. When TX-line and RX-line are referred, these are the lines looked at from the perspective of the debugging hardware. The communication topology of host-PC-to-debugging-hardware is shown in Figure 4.1.

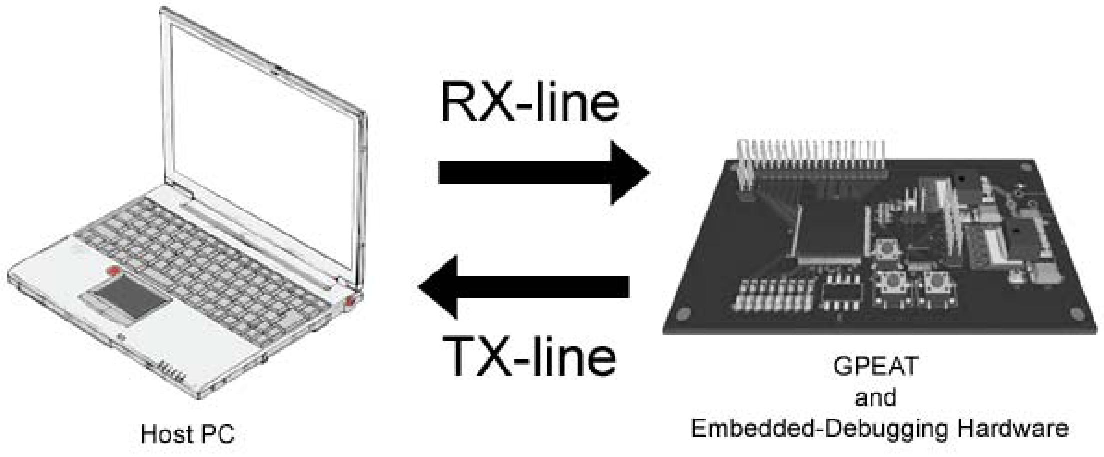
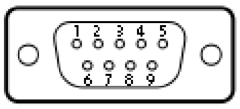


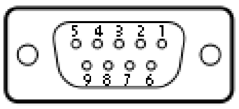
Figure 4.1 Communication topology

4.3.1 RS-232 Connector

The connectors used in RS-232 connectors used here are D-subminiature with 9-pin, called DB9. The front view of a DB9 Male connector (DB9M) and a DB9 Female connector are shown in Figure 4.2.



(a) DB9M connector (Male)



(b) DB9F connector (Female)

Figure 4.2 front view of DB9 connector

Male connectors (DB9M) are the terminals out from a PC. The GPEAT debugging hardware will use a DB9 Female connectors (DB9F). The link cable is made up with two-conductor shielded cable which each end of the cable is connected with DB9F. The signal-pin mapping here follows the EIA-232 standard which is shown below in Table 4.1.

Pin	Signal	Pin	Signal
1	CD – Carrier Detect	6	DSR – Data Set Ready
2	RXD – Received Data (RX-line)	7	RTS – Request To Send
3	TXD – Transmitted Data (TX-line)	8	CTS – Clear To Send

4	DTR – Data Terminal Ready	9	RI – Ring Indicator
5	GND – Signal Ground		

Table 4.1 Signal-Pin mapping in RS-232 connection

There are many communication schemes for connecting cables using an RS-232 connection. In this literature, the scheme called *Null-Modem* is used which is simple and does not use some of the handshaking methods used in other RS-232 communication schemes. In null-modem wiring scheme, the TX-line at one end is connected to RX-line of the other end. RTS and CTS is directly connected together because we are not using a data-flow-control handshake. GND terminals are of the two ends are connected together for signal reference. Null-Modem cable wiring is shown in Figure 4.3.

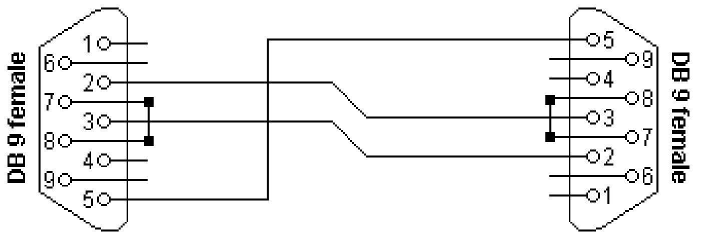


Figure 4.3 Null-Modem cable wiring

4.3.2 RS-232 Universal Asynchronous Receiver/Transmitter (UART) Protocol

Data sent and received through the RS-232 connection is divided into frames. A frame is a non-divisible packet of bits. It consists of both data byte (information) and overhead (start-bit and stop-bit). One frame is made up of one start-bit, seven bits of information, one parity-bit and one stop-bit in accordance with UART protocol. If a frame was sent previously and next frame is about to be sent immediately without delay, the stop-bit of the previous frame will be replaced with the start-bit of the next frame. The protocol used in this literature does not use the parity-bit and the parity bit will not be used for error checking. The parameters for RS-232 protocol used are summarized in Table 4.2 and a timing diagram for a frame is shown in Figure 4.4.

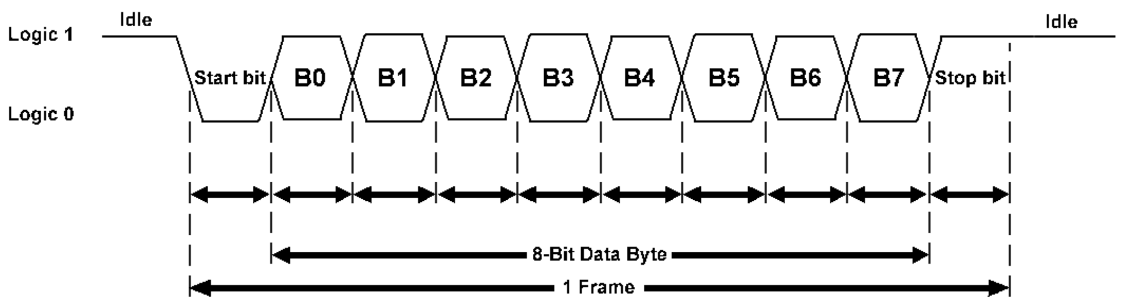


Figure 4.4 Timing diagram for a frame transferred in RS-232 connection

Comm. Port Parameters	**Value**	**Unit**
Data bits	8	Bits
Stop bit	1	Bit
Parity bit	None	
Flow Control	None	
DTR line	Not applicable	
RTS line	Not applicable	

Table 4.2 Parameters for RS-232 protocol

4.3.3 USB-to-RS232 Converter

Many computers today especially most of laptops do not provide an RS-232 port. Because of this we used a port converter in order to connect an existing computer port to the board's RS-232 port. The USB-to-RS232 converter is an ideal solution for this. It can be recognized as a built-in RS-232 port in certain operating systems and is able to work according to EIA-232 standard and, therefore, does not need extra programming on the computer side to send and receive UART formatted data. In addition, a USB port has very high-speed data transfer rate of 400Mbps which is an advantage for a converted RS232 port. Normally, RS232 port has a limit for data transfer rate of 57600 bps but with our configuration the port can transfer data at a rate up to 921600 bps. By using the transfer rate of 921600 bps, the bit-error-rate (BER) is high due to the timing problem caused by the clock speed used in GPEAT system which is 50MHz. The feasible data transfer rate for RS-232 connection used in this literature is 460800 bps which is still faster than the normal rate. Figure 4.5 shows a USB-to-RS232 converter.

Figure 4.5 A USB-to-RS232 converter

4.4 Hardware Architecture and Implementation

The debugging hardware is implemented as an add-on component for the GPEAT system. It obtains all information on every individual in each population and sends this information back to the host PC through the RS-232 connection. The debugging hardware consists of 10 main modules. Five of these modules are implemented in finite state machines (FSMs). The FSMs in these designs generate control signals and monitor for notifying signal to interact among the modules. These controllers are organized in a manner similar to a microcontroller. The main unit (debugger controller [DC]) performs like a CPU which controls its peripheral controllers and interacts with the memory unit. The transmitting controller (TXC) and receiving controller (RXC) is responsible for RS-232 communications and acts like peripheral controllers of the microcontroller. The DC provides control signals to TXC, RXC and the memory unit (RAM). It also interacts with the GPEAT system which in turn can be considered as a controlling system for the DC. The DC can be considered as a peripheral controller from the perspective of the GPEAT system.

In the following sections, we limit our interest to the hardware modules of the debugger. Again, the debugger is a component that communicates with the GPEAT system in order to capture the values being used in the GPEAT system. We describe the hardware modules and illustrate the ASM charts for each controller.

4.4.1 Overall Hardware Design

We introduce the overall hardware design in this section and describe the high-level hardware interactions and specifications. Figure 4.6 shows the hierarchical diagram of controller interactions.

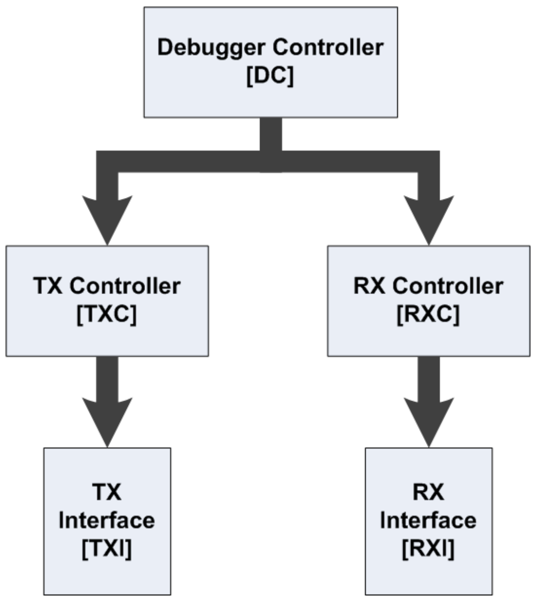


Figure 4.6 Hierarchical diagram for modules

Five FSMs are in charge of controlling data flow from GPEAT to the host PC. The DC unit accepts debugging information, such as chromosome size, gene size, chromosome content and fitness value from GPEAT, and sends this information back to host PC as bytes package through RS-232 connection using TXC and TXI. There is no information that is sent from the host PC to GPEAT. RXC and RXI are only responsible for receiving commands from host PC to the DC unit.

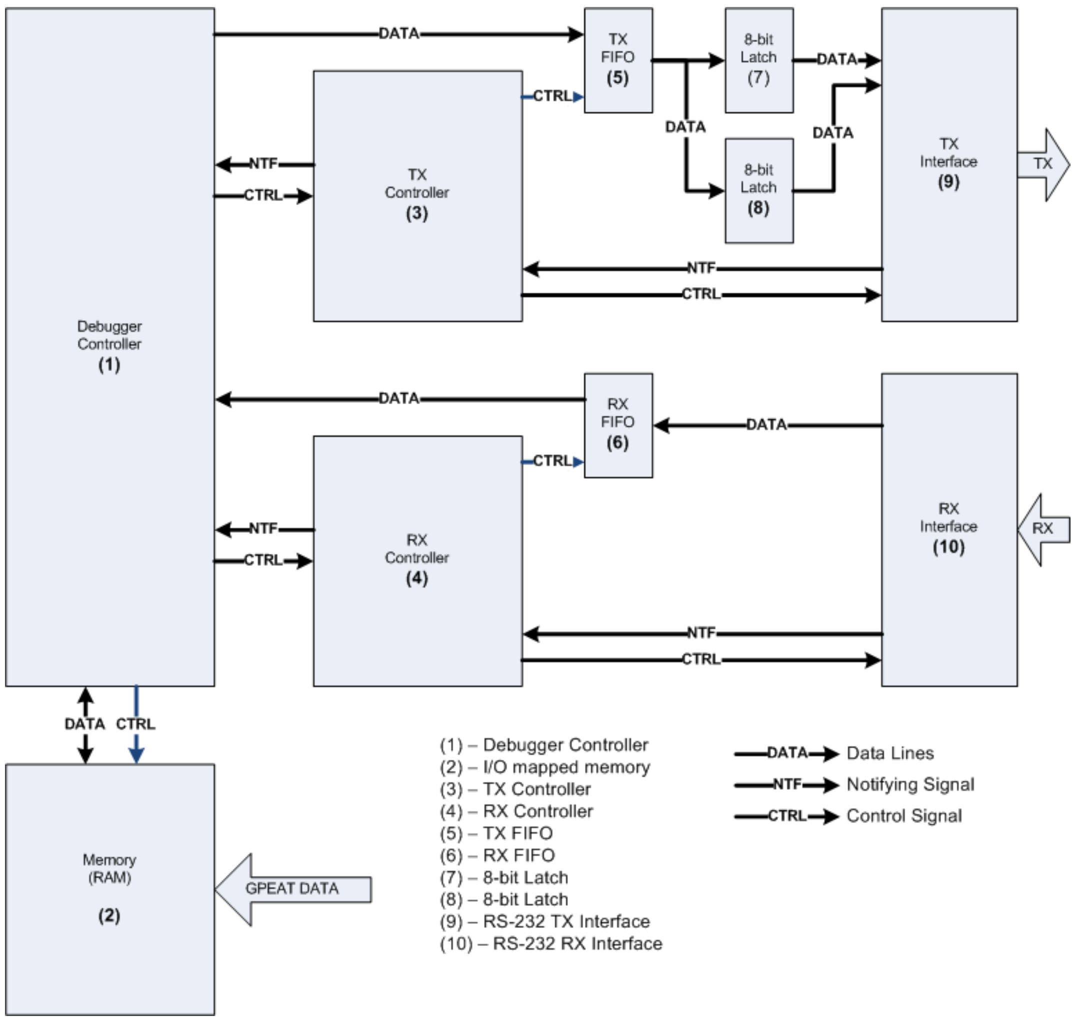


Figure 4.7 Overall architecture for debugging hardware

Figure 4.7 shows the high-level block diagram describing the interactions among hardware modules. Modules (1), (3), and (4) are controllers whereas (9) and (10) are line-interfaces. All of the controllers and interfaces are FSMs whose behaviors can be described using ASM charts. The ASM charts are explained in details in the following sections (4.4.2 – 4.4.7). There are some hardware modules which are not FSMs. These blocks include latches for data being processed and three memory modules for data-buffering purpose.

The DC provides control signals to TXC and RXC and monitors for notifying signals from those controllers. When GPEAT has some information to send back, the DC checks if TXC is busy or not. If TXC is not busy it interacts with TXC by sending data accepted from GPEAT core to FIFO of TXC and designates TXC to initiate a sending transaction through TXI.

Whenever RXC receives a command (a data byte) from the host PC through RXI, it stores the command in its FIFO making a queue of commands to be sequentially executed by the DC. RXC notifies the DC when its FIFO is not empty. After a notification the DC will strobe the commands out from the FIFO one at a time. Normally commands will not be accumulated in the FIFO of RXC since RXC notifies the DC every time a single command is received but the concept of queuing commands in the FIFO is important in that it is the mechanism for preventing loss of commands if the DC cannot completely execute commands before the next command arrives.

TXI and RXI are FSMs that communicate with the host PC through the RS-232 TX- and RX-line. These FSMs have to meet the timing requirements for the baud rates defined in the EIA-232 standard. When a command is received, RXI requests RXC to stores this command in the FIFO and, when a data package needs to be sent, TXC requests TXI to send the package one byte at a time.

4.4.2 The Debugger Controller (DC)

The DC controls TXC and RXC directly and receives/sends data packages for the RS-232 connection through these controllers. TXC and RXC provides a *ready* signal for the DC in order to indicate that they are ready and are available to interact with the DC unit.

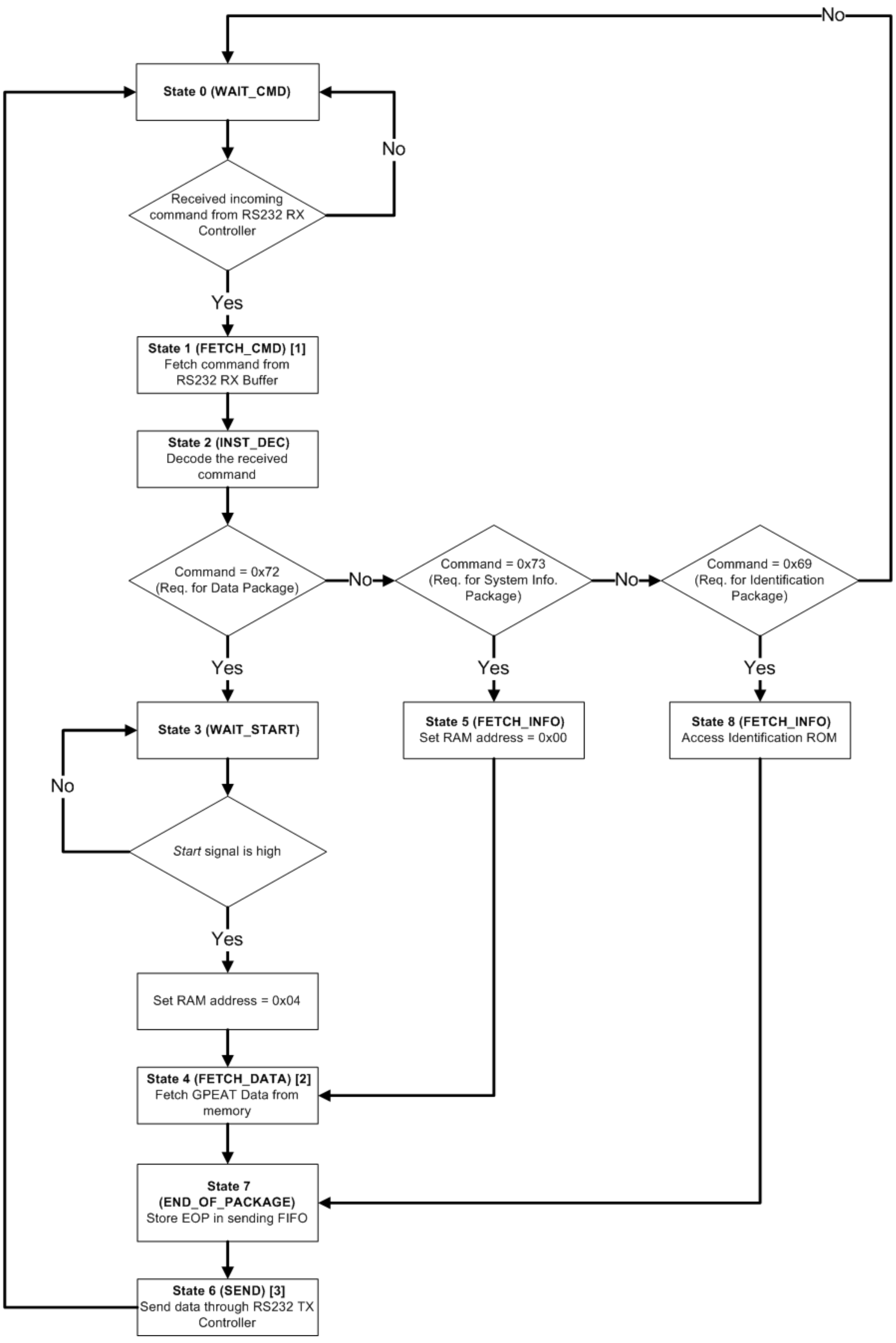

State 0 (WAIT_CMD)
Received incoming command from RS232 RX Controller
No
Yes
State 1 (FETCH_CMD) [1] Fetch command from RS232 RX Buffer
State 2 (INST_DEC) Decode the received command
Command = 0x72 (Req. for Data Package)
No
Command = 0x73 (Req. for System Info. Package)
No
Command = 0x69 (Req. for Identification Package)
No
Yes
Yes
Yes
State 3 (WAIT_START)
State 5 (FETCH_INFO) Set RAM address = 0x00
State 8 (FETCH_INFO) Access Identification ROM
No
Start signal is high
Yes
Set RAM address = 0x04
State 4 (FETCH_DATA) [2] Fetch GPEAT Data from memory
State 7 (END_OF_PACKAGE) Store EOP in sending FIFO
State 6 (SEND) [3] Send data through RS232 TX Controller

Figure 4.8 The ASM chart for the DC unit

The Finite State Machine (FSM) for the DC unit has six states for carrying out the processes of receiving commands and sending data back through RS-232 connection to and from the computer. In state_0 (WAIT_CMD), the controller waits for an incoming command sent by the host PC. The controller will stay idle as long as it is not receiving a command from the host PC. After the RXC signifies to the DC that its FIFO is not empty, the DC immediately proceeds to state_1 (FETCH_CMD) to interact with RXC by changing the mode of RXC to the (receiving mode)—modes of RXC will be described in section 4.4.5 (RX Controller). When RXC is in receiving mode, the DC strobes the command or data out from the RXC FIFO by providing a data-strobe signal to RXC. Two clock cycles after data-strobe has been received, RXC presents valid data on its 8-bit data terminal to the DC. The presented command is obtained by the DC and then decoded in state_2 (INST_DEC) of the DC. Three commands are recognized by the DC and are summarized in Table 4.3. The command format for the DC is a bit-string with a length of eight bits (recognized as a byte).

Command	**Description**
0x72 (‘r’ in ASCII)	Request for Chromosome Package
0x73 (‘s’ in ASCII)	Request for System Information. Package
0x69 (‘i’ in ASCII)	Request for Identification Package

Table 4.3 Command recognized by the DC

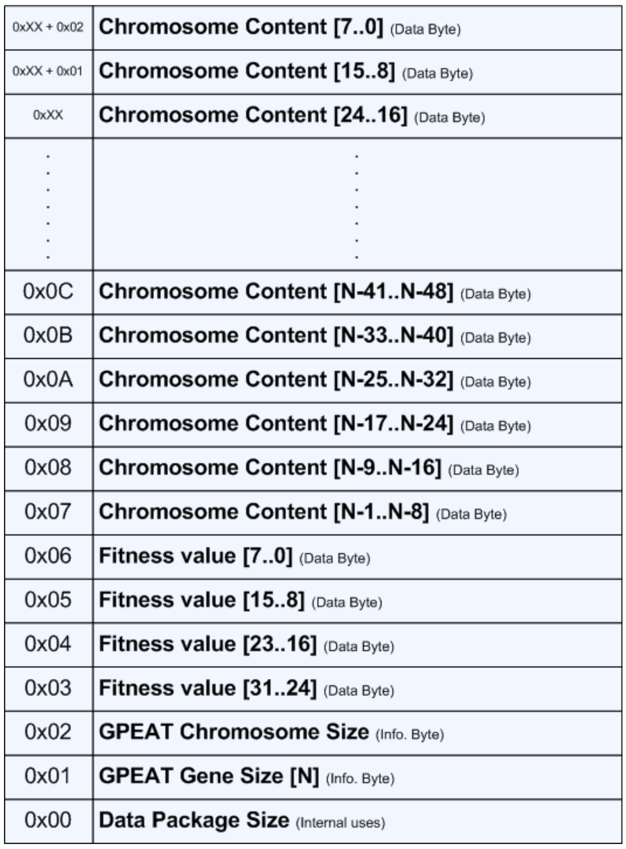
When the DC receives a 0x72 command (sending data), the controller proceeds into next state, state_3 (WAIT_START). The DC waits for *start* signal from GPEAT system in this state. The GPEAT system has to present the binary content of a chromosome to be sent and a 32-bit binary value of that particular chromosome to the DC then the debugger is called by raising the *start* signal high for two clock cycles. After finishing this initialization for sending a chromosome and its fitness value back to the computer, the DC fetches all data in its RAM and sends that data back in state_4 (FETCH_DATA) and state_6 (SEND). A chromosome and its fitness value given to the DC will be stored into the RAM of the DC. This mechanism is explained in the following section 4.4.2 – Memory (RAM).

The 0x73 command (request to send system information) informs the DC to send information about the system back to the host PC. This system information package consists of chromosome size, gene size, gene type and population. When the host PC receives this package, it recognizes the received data in the domain of EC and treats the data as GPEAT does. When the DC receives this command, it immediately steps through to state_6 (SEND) and sends the information package back. This is sent immediately since this information has already been defined when the GPEAT system was synthesized.

The 0x69 command notifies the DC to send an identification package back to the computer. An identification package contains the hardware signature and GPEAT version and will be recognized by the debugging GUI running on the host PC and used to verify the debugging hardware. When the DC receives this command it accesses an identification ROM which can only be altered by the GPEAT designers.

4.4.3 Memory (RAM)

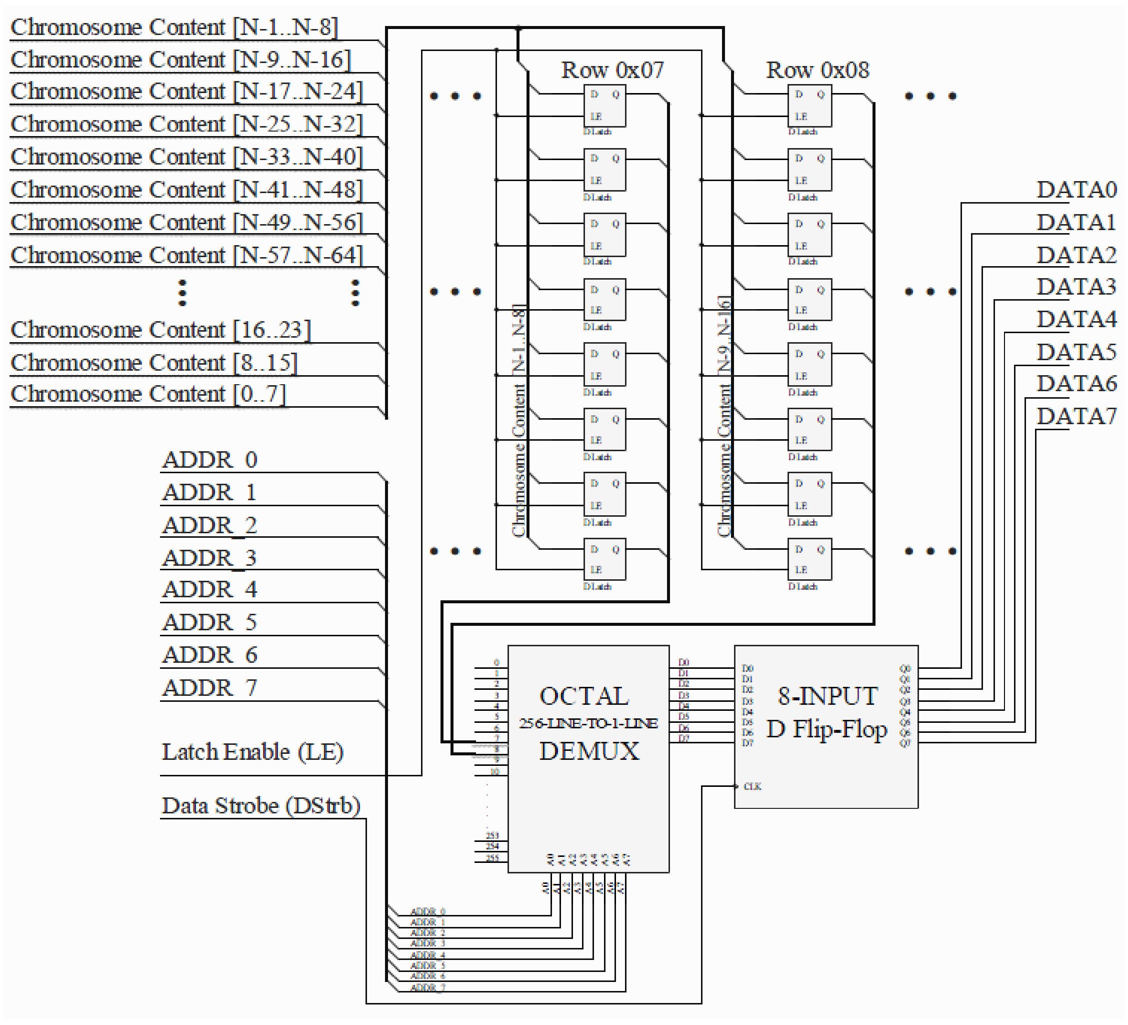
The memory addressing model used here is a linear address space model where all the memory address can be expressed completely with no offset, by a value stored in a single register. This memory module is accessed by the DC as a RAM module. The DC uses an 8-bit register, a BI (Byte Indexing register), to address the RAM which contains both information packages and data packages. Data mapping is illustrated in Figure 4.4.



0xXX + 0x02	**Chromosome Content [7..0]** (Data Byte)
0xXX + 0x01	**Chromosome Content [15..8]** (Data Byte)
0xXX	**Chromosome Content [24..16]** (Data Byte)
⋮	⋮
0x0C	**Chromosome Content [N-41..N-48]** (Data Byte)
0x0B	**Chromosome Content [N-33..N-40]** (Data Byte)
0x0A	**Chromosome Content [N-25..N-32]** (Data Byte)
0x09	**Chromosome Content [N-17..N-24]** (Data Byte)
0x08	**Chromosome Content [N-9..N-16]** (Data Byte)
0x07	**Chromosome Content [N-1..N-8]** (Data Byte)
0x06	**Fitness value [7..0]** (Data Byte)
0x05	**Fitness value [15..8]** (Data Byte)
0x04	**Fitness value [23..16]** (Data Byte)
0x03	**Fitness value [31..24]** (Data Byte)
0x02	**GPEAT Chromosome Size** (Info. Byte)
0x01	**GPEAT Gene Size [N]** (Info. Byte)
0x00	**Data Package Size** (Internal uses)

Figure 4.9 Data mapped in the RAM module

Figure 4.4 shows how data packages are mapped into the 256 x 8 bits RAM. The DC accesses address 0x00 for internal uses and accesses addresses 0x01 and 0x02 for content for the information package which contains GPEAT gene size and chromosome size. Addresses from 0x03 to 0x06 store fitness values for chromosome. Addresses beyond address 0x06, namely 0x07 to 0xFF, are reserved space for storing chromosome content. The chromosome content area can store up to 1992 bits (249 multiplied by 8).



Chromosome Content [N-1..N-8]
Chromosome Content [N-9..N-16]
Chromosome Content [N-17..N-24]
Chromosome Content [N-25..N-32]
Chromosome Content [N-33..N-40]
Chromosome Content [N-41..N-48]
Chromosome Content [N-49..N-56]
Chromosome Content [N-57..N-64]
Chromosome Content [16..23]
Chromosome Content [8..15]
Chromosome Content [0..7]
ADDR 0
ADDR 1
ADDR 2
ADDR 3
ADDR 4
ADDR 5
ADDR 6
ADDR 7
Latch Enable (LE)
Data Strobe (DStrb)
Row 0x07
Row 0x08
Chromosome Content [N-1..N-8]
Chromosome Content [N-9..N-16]
D Latch
OCTAL
256-LINE-TO-1-LINE
DEMUX
8-INPUT
D Flip-Flop
CLK
DATA0
DATA1
DATA2
DATA3
DATA4
DATA5
DATA6
DATA7

Figure 4.10 RAM architecture for the DC

Figure 4.10 shows the architecture for the RAM module for the DC. Note that the diagram only partially shows the required elements in the RAM module—and does not entirely shows what is inside the RAM modules. This diagram describes the low-level signal connections which can be used to control the behavior of the RAM module.

We start the explanation of the RAM module with the data latching operation of the RAM—which is similar to write operation of commonly used RAM. The DC presents chromosome content and fitness value in bits through *Chromosome Content* and *Fitness Value* terminals of the RAM while it holds the *Latch Enable (LE)* signal high. When the LE signal is high, the values in the D-Latch arrays change simultaneously with the chromosome content and Fitness value data. The chromosome content and Fitness value data is presented by the DC. The DC makes sure that the data to be sent is not changed by keeping the LE signal low. In this case, the data is latched and stored in a D-Latch arrays. The data is held and will not change even when the data presented at the RAM terminals is changed. The data at the RAM terminal is ignored and will not be stored in D-Latch arrays as long as the LE signal is low.

The read operation is performed by changing the LE signal to low. In this case too, the D-Latch arrays hold the data as explained above. Then the DC asserts the address to be read at the address terminals of the RAM. A triggering signal, namely *Data-Strobe (DStrb)*, is raised to high by the DC to signify to the RAM module to change the data value at the *Data-out* terminals according to the address presented at the RAM terminals.

To understand how the RAM is accessed by the DC, we previously introduce the BI register in the DC. Here we will introduce another 8-bit register, the PS (Package Size register), which is associated with the BI register and contains the number of bytes to send in one transaction. The PS register will be loaded with either a value of 0x04 or the value stored in address 0x00 depending on whether the DC is operating command 0x73 or 0x72. For command 0x73, the DC sends a system information package back to host PC. In this case the PS register is loaded with value 0x04. If the host PC requests for a data package by sending command 0x72 to the DC, the PS register will automatically be loaded with the value stored in the RAM at address 0x00 (Data Package Size). This value depends on the size of chromosome in the GPEAT system and the value can be determined by the size of chromosome in bits divided by eight. If there is a remainder when the chromosome size has been divided by eight, the data package size is rounded up to the next number of bytes so all bits can be accommodated. For both commands, the DC has an internal mechanism that automatically loads the size of package into the PS register. No command needed to direct this action. This is described in the ASM chart below in Figure 4.11.

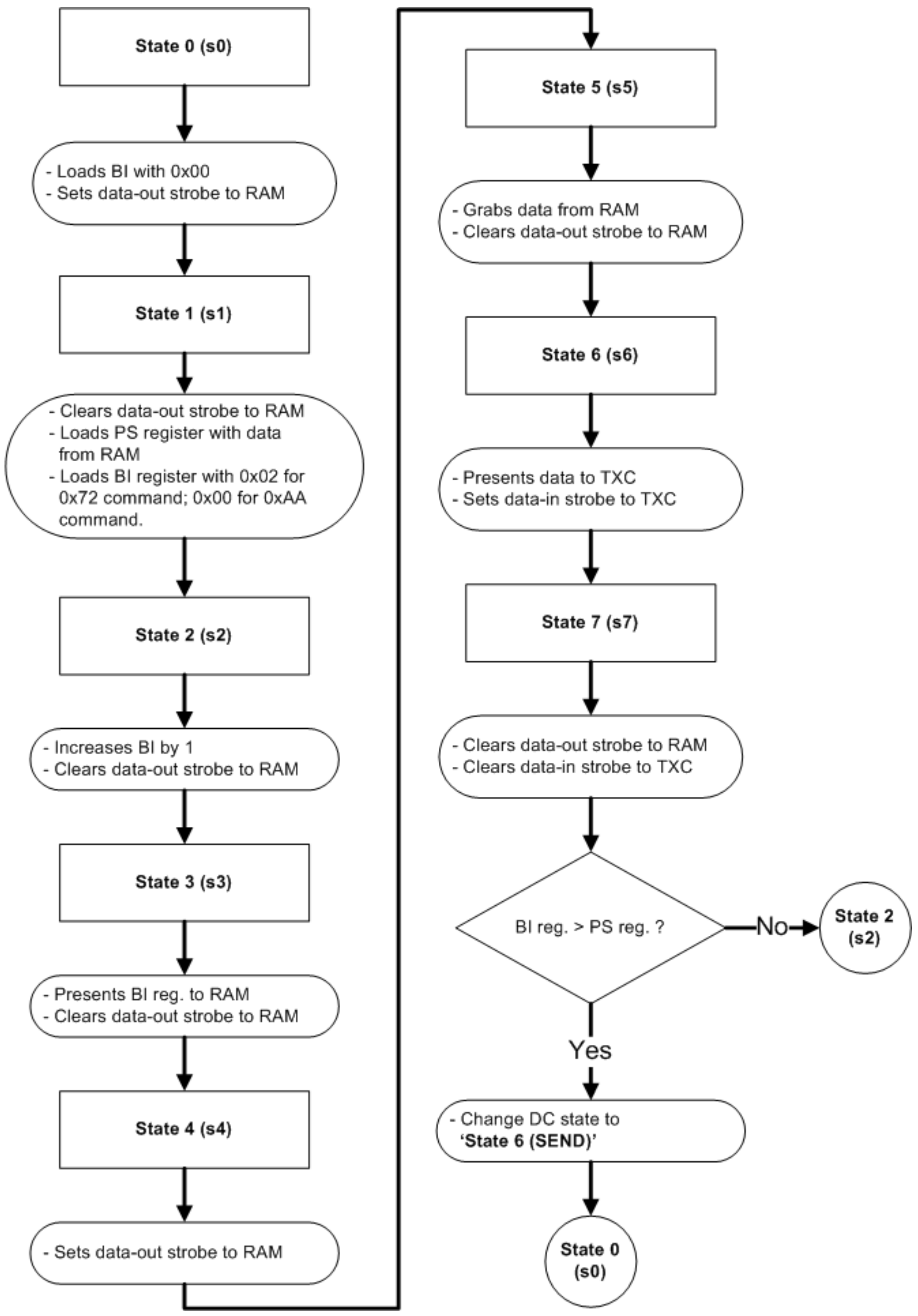

State 0 (s0)
- Loads BI with 0x00
- Sets data-out strobe to RAM
State 1 (s1)
- Clears data-out strobe to RAM
- Loads PS register with data from RAM
- Loads BI register with 0x02 for 0x72 command; 0x00 for 0xAA command.
State 2 (s2)
- Increases BI by 1
- Clears data-out strobe to RAM
State 3 (s3)
- Presents BI reg. to RAM
- Clears data-out strobe to RAM
State 4 (s4)
- Sets data-out strobe to RAM
State 5 (s5)
- Grabs data from RAM
- Clears data-out strobe to RAM
State 6 (s6)
- Presents data to TXC
- Sets data-in strobe to TXC
State 7 (s7)
- Clears data-out strobe to RAM
- Clears data-in strobe to TXC
BI reg. > PS reg. ?
No
State 2 (s2)
Yes
- Change DC state to 'State 6 (SEND)'
State 0 (s0)

Figure 4.11 The ASM chart for sub-state in state 'FETCH_DATA' of the DC

Note that these states described in Figure 4.11 are the sub-states which are performed in state 4 -'FETCH_DATA' of the DC. The DC starts fetching data by obtaining the obtain the size of data package by accessing the value stored at the address 0x00 of the RAM. It loads the PS register with 0x04 if the received command is 0x73 and loads the value stored in the RAM location 0x00 into the PS register if the received command is 0x72. The next step is to increase the BI register by 1 and present the value stored in the BI register to the RAM so that the next byte will be read. This step (state 2) through the final step (state 7) will be repeated until the DC reads the number of bytes specified in PS register. The next step is state 4 and state 5 where the DC grabs the data presented at the data-out terminals of the RAM. After obtaining data from the RAM, it is necessary for the DC to buffer the data in TXC by requesting TXC to store each individual byte in the TXC FIFO in state 6 and state 7. After finishing all sub-state (s0-s7) in state 4 (FETCHING_DATA), the DC moves on to the next state which is state 6 (SEND) which tells TXC to send the data stored in TXC FIFO.

4.4.4 The TX Controller (TXC)

The tasks for TXC is to communicate and exchange data with the FIFO. This includes controlling TXI for sending data back to host PC. TXC operates in two modes which are *TX Interface Controller Buffering (Mode-0)* and *TX Interface Controller Sending (Mode-1)*. The DC determines the mode for TXC and changes the mode. Figure 4.12 shows ASM charts for TXC in each mode.

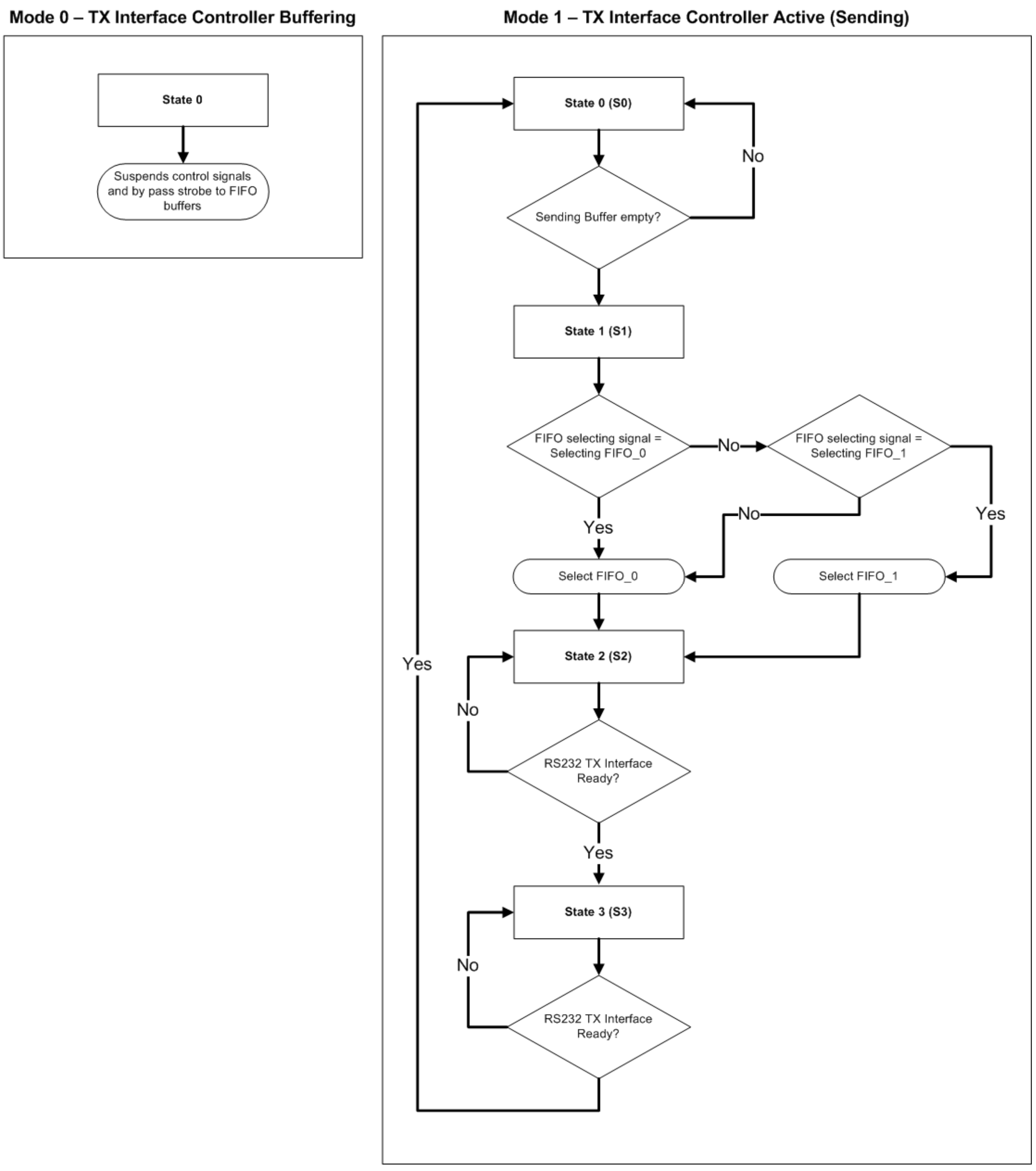

Mode 0 – TX Interface Controller Buffering
State 0
Suspends control signals and by pass strobe to FIFO buffers
Mode 1 – TX Interface Controller Active (Sending)
State 0 (S0)
No
Sending Buffer empty?
State 1 (S1)
FIFO selecting signal = Selecting FIFO_0
No
FIFO selecting signal = Selecting FIFO_1
No
Yes
Yes
Select FIFO_0
Select FIFO_1
State 2 (S2)
Yes
No
RS232 TX Interface Ready?
Yes
State 3 (S3)
No
RS232 TX Interface Ready?

Figure 4.12 The ASM charts for TXC in mode-0 and mode-1

The FIFO can be considered as a built-in unit in the TXC block. The FIFO will receive control signals directly from either the DC or TXC. In mode-0, which is described by Figure 4.12 (mode-0), TXC suspends all control signals to FIFO except for the R/W signals, the signals which control whether the FIFO will receive or providing data. The R/W signals in mode-0 will be cleared to low indicating that the FIFO will receive data directly from the DC. TXC also bypasses the data-in strobe signal that it receives from the DC to the FIFO in this mode.

TXC will be active in mode-1. All control signals are overridden by TXC in this mode; and the DC will no longer be able to control the FIFO. In this mode, TXC controls the FIFO and TXI directly and is responsible for obtaining data stored in FIFO (previously stored in mode-0) and sending this data to TXI.

As shown in Figure 4.12 (mode-1), TXC starts the process in state-0 (s0) by observing the *empty* signal from the FIFO. This signal indicates whether there is data left in the FIFO or not. If this signal is high, TXC provides a data-out strobe signal to the FIFO and obtains data byte one-at-a-time. There are two sets of data-in terminals from the FIFO to TXC. In this state, only one set of data-in terminals will be used to obtain data from FIFO. TXC determines which set of terminals will be active and it toggles the status (active/inactive) of these sets of terminals after each use. The purpose of having two sets of data-in terminals from the FIFO is that TXC can receive data bytes from the FIFO continuously even when TXI is busy. When TXI is available again, TXC has already obtained next data byte from the FIFO and this data byte will be immediately presented at the other set terminals of TXI and sent through TXI when TXI becomes available.

4.4.5 The TX Interface (TXI)

TXI is the FSM that responsible for sending data byte through RS-232 connection. TXI sends data bytes accordingly to the UART protocol. The parameters for RS-232 connection specified in TXI are summarized below in Table 4.4.

Comm. Port Parameters	**Value**	**Unit**
Data bits	8	Bits
Stop bit	1	Bit
Parity bit	None	
Flow Control	None	
DTR line	Not applicable	
RTS line	Not applicable	
Baud rate	110, 300, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600	Bps

Table 4.4 Parameters for RS-232 protocol in TXI

TXI has a built-in baud-rate-generator (BRG) which produces the clock signal. TXI uses this baud clock for sending data back to host PC and it also uses this clock signal for its FSM. This BRG continuously generates the clock signal with clock speed according to baud-rate value in Table 4.4. This is in contrast with the BRG in RXI which will be explained later.

TXC divides the data package or information packages into bytes then sends each byte to TXI. TXI receives one byte from the TXC FIFO at a time and sends it along with a start- and stop-bit. TXI starts each transaction by clearing the TX-line to low first for one bit worth of data (the start bit). This indicates to the host PC to be ready to receive data. Bits following start-bit are eight data bits and then a stop-bit. After TXI successfully the sends start-bit, it immediately changes the state of TX-line accordingly to data bits and then sets TX-line to high for the final stop bit which effectively ends the transaction. This process can be illustrated in details in Figure 4.8.

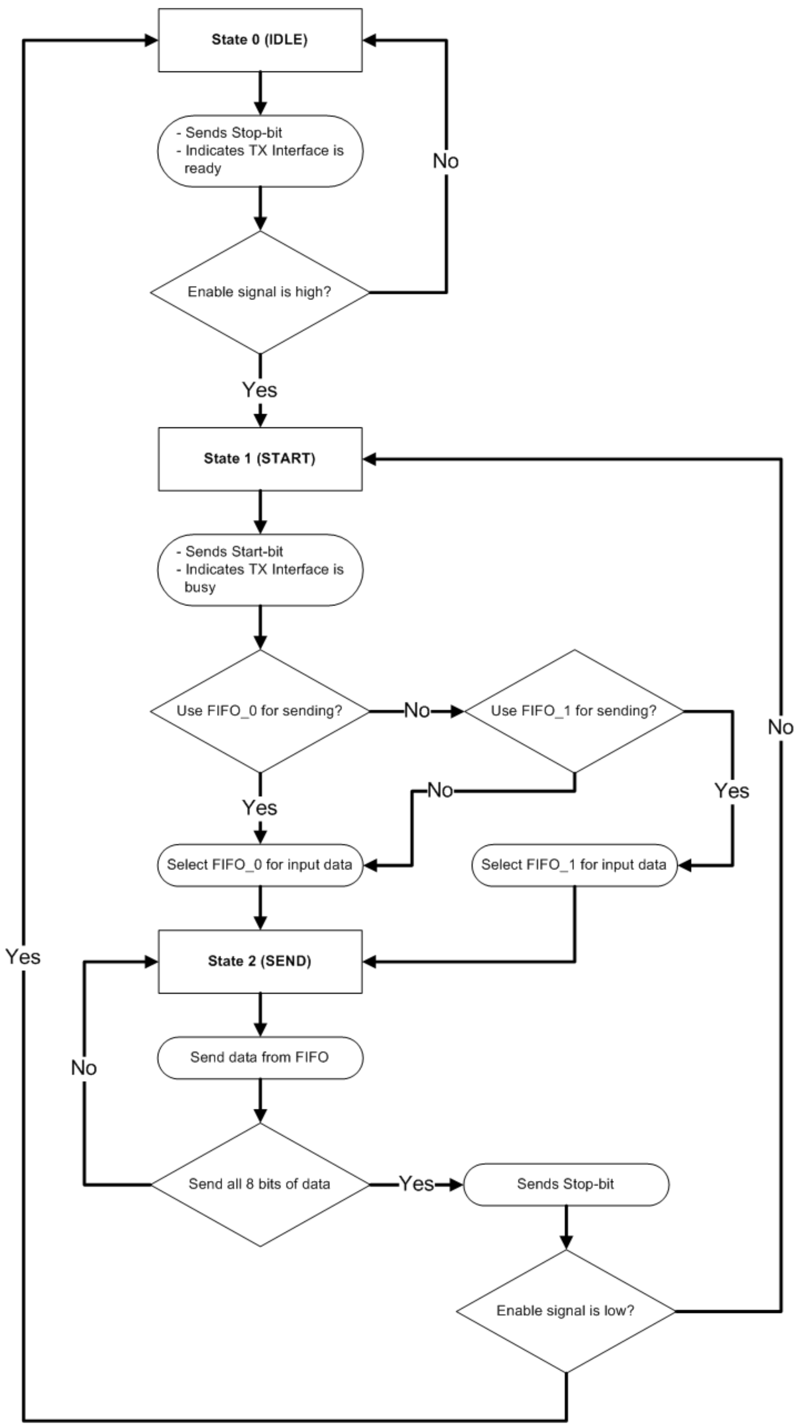


Figure 4.13 The ASM chart for TXI

According to the ASM chart in Figure 4.13, TXI continuously sends the stop-bit value through TX-line to host PC in state-0 (IDLE) as well as setting the *busy* signal low indicating that it is available to the TXC. Simultaneously, it monitors for the *enable* signals from TXC. If the *enable* signal is set to high by TXC, TXI changes the state to state-1 (START) and starts sending the start-bit. In state-1 (START), TXI holds the TX-line to the state of start-bit for one baud-clock cycle and indicates to the TXC that it is busy by raising busy signal to high. Two sets of data-in terminals, previously mentioned in section 4.4.4, are used again here in state-1 (START). One of these two sets of terminals is selected by TXC to obtain data byte from the TXC FIFO. TXC provides a *data-in terminal select* signal to choose which set of terminals will be used. If data-in terminal select signal is high, terminal-1 will be used to obtain data from TXC FIFO. Otherwise terminal-0 is used. After obtaining data from the TXC FIFO, TXI proceeds to next state, state-2 (SEND), to send the data byte grabbed. State-2 (SEND) will be repeated nine times because every time state-2 is repeated TXI sends only one bit of data byte and, at the ninth iteration, the stop-bit is sent and the *enable* signal is observed again. If the *enable* signal is high, TXI goes back to state-1 (START) again immediately sending the start-bit for the next byte that will also come from TXC FIFO. If the *enable* signal is low, TXI goes back to state-0 (IDLE) and waits to be reactivated once *enable* signal goes high again.

4.4.6 The RX Controller (RXC)

RXC is the controller responsible for receiving data byte (command) from host PC. RXC controls the RXC FIFO and RXI for receiving and buffering data for the DC. The DC obtains command stored in the RXC FIFO when RXC notifies the DC that one or more commands is available in its FIFO. The two operation modes and ASM charts for RXC are shown in Figure 4.14.

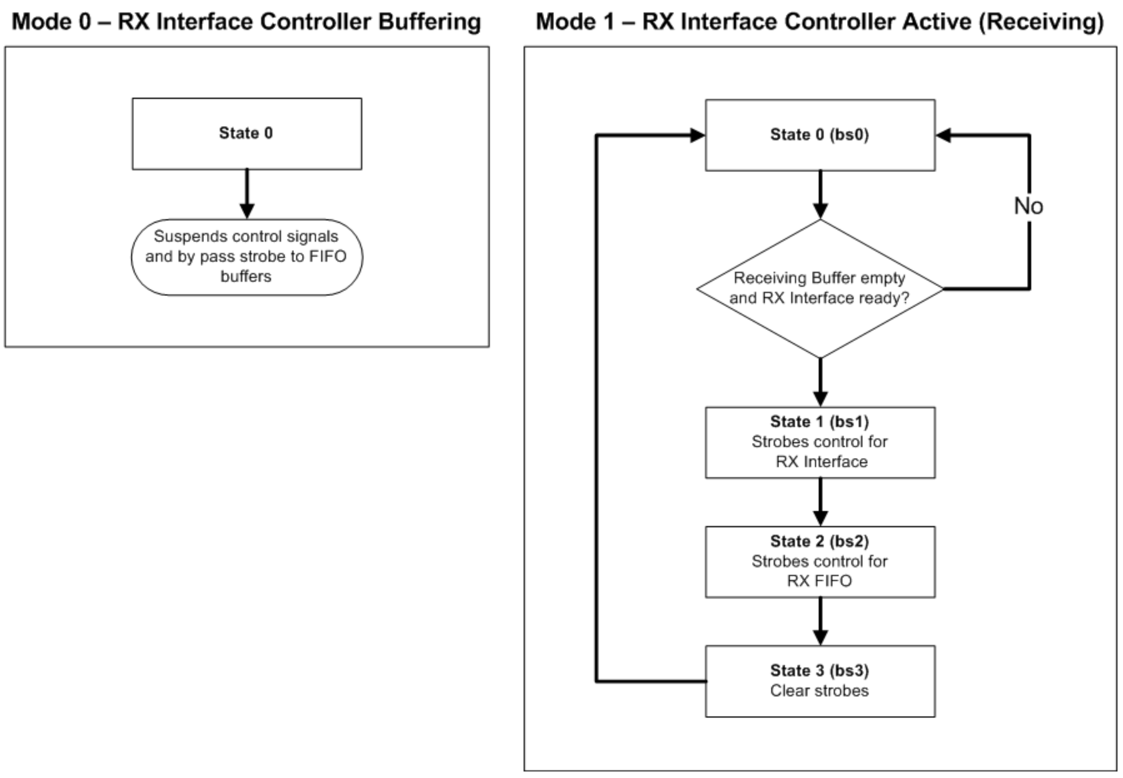

Mode 0 – RX Interface Controller Buffering
State 0
Suspends control signals and by pass strobe to FIFO buffers
Mode 1 – RX Interface Controller Active (Receiving)
State 0 (bs0)
No
Receiving Buffer empty and RX Interface ready?
State 1 (bs1) Strobes control for RX Interface
State 2 (bs2) Strobes control for RX FIFO
State 3 (bs3) Clear strobes

Figure 4.14 The ASM charts for RXC

Similar to those in TXC, RXC also has two operation modes and they are *mode-0 (RX Interface Controller Buffering)* and *mode-1 (RX Interface Controller Active)*. In mode-0, the DC sends control signals to RXC to obtain the commands stored in the FIFO. RXC suspends all control signals except the R/W signals to the FIFO. The R/W signal is set to high to indicate to the FIFO that is should present the data stored when it receives a data-out strobe. The data-out strobe signal from the DC is directly fed to the FIFO because RXC bypasses this signal and allows the DC to control when to clock the data out from the FIFO. In mode-1, RXC is active and that means it is receiving commands from the host PC and storing the commands in the FIFO. The two main steps for receiving a command from host PC are the 'monitoring RXI/obtain command if RXI has one' and the 'buffering command in the FIFO'. For hardware construction reasons, these two main steps are distributed into four states as shown in the ASM chart (mode-1). First, RXC monitors for a *busy* signal from RXI in the state-0 (bs0). If RXI is busy it will wait in state-0, otherwise it moves to state-1 (bs1). RXC raises data-out strobe signal to RXI to high for obtaining the received command from RXI then it moves to state-2 (bs2). RXC stores the received command from RXI in the FIFO by raising data-in strobe signal to the FIFO to high. After storing the received command it moves to state-3 (bs3). In state-3, RXC clears all strobe signals to low and goes back to state-0 to wait for another command to be received.

4.4.7 The RX Interface (RXI)

RXI is the FSM that is responsible for receiving data bytes from host PC. RXI receives data bytes accordingly to the UART protocol. The parameters for RS-232 connection specified in TXI are summarized below in Table 4.5.

Comm. Port Parameters	**Value**	**Unit**
Data bits	8	Bits
Stop bit	1	Bit
Parity bit	None	
Flow Control	None	
DTR line	Not applicable	
RTS line	Not applicable	
Baud rate	110, 300, 1200, 2400, 4800, 9600, 19200,	Bps

	38400, 57600, 115200, 230400, 460800, 921600	

Table 4.5 Parameters for RS-232 protocol in RXI

TXI has previously been described in section 4.4.5. It sends data bytes to host PC through the TX-line. The same process occurs here but in the reversed order. We will now explain the reversed process in which host PC sends data to RXI through RX-line. The RXI unit has a built-in baud-rate-generator (BRG) block similar to the TXI's. The difference is that the BRG in RXI will only be active when RXI receives a start-bit and will be inactive while RXI receives a stop-bit. When the BRG is active, it provides a baud clock to RXI. Figure 4.15 shows the ASM charts for both the BRG (on the left) and RXI (on the right).

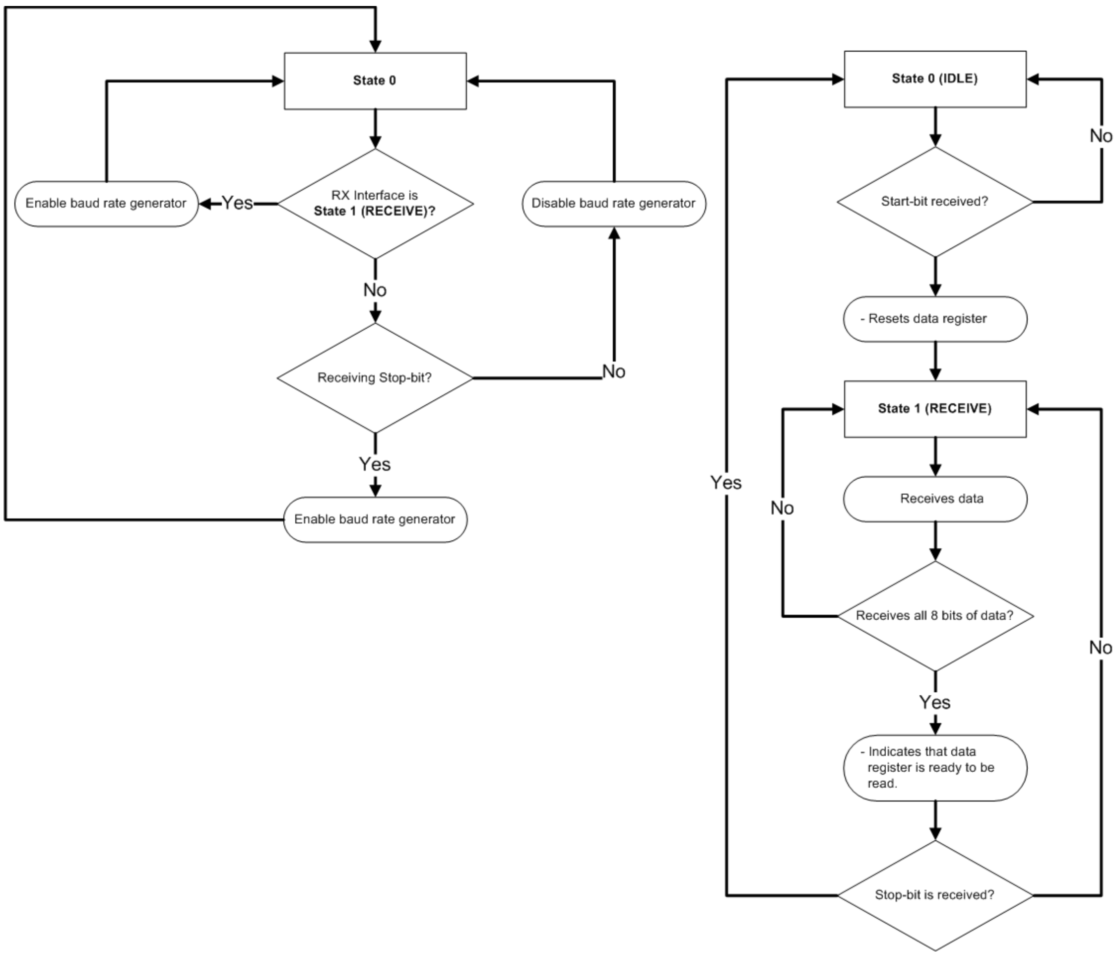

State 0
Enable baud rate generator
Yes
RX Interface is State 1 (RECEIVE)?
Disable baud rate generator
No
Receiving Stop-bit?
No
Yes
Enable baud rate generator
State 0 (IDLE)
No
Start-bit received?
- Resets data register
State 1 (RECEIVE)
Yes
No
Receives data
Receives all 8 bits of data?
No
Yes
- Indicates that data register is ready to be read.
Stop-bit is received?

Figure 4.15 The ASM charts for BRG (left) and RXI (right)

First, we explain the ASM chart for the BRG on the left of Figure 4.15. When BRG is in active, it checks the state of RXI. If RXI is in state-1 (RECEIVE), it enables baud clock circuitry to provide the clock signal for RXI. Otherwise it monitors for a stop-bit. If a stop-bit is received, it disables the baud clock circuitry and stops feeding the clock signal to RXI.

The ASM chart for RXI on the right has two states for monitoring and receiving command from host PC. In state-0 (IDLE), RXI monitors for a start-bit on RX-line. Note that at this time RXI is not receiving the clock signal from the BRG. The consequence is that RXI cannot change the state to state-1 (RECEIVE), when BRG is in active. As soon as BRG captures a start-bit and becomes active, RXI receives the clock signal from BRG and is able to change the state to state-1. RXI receives all the bits in a command by repeating state-1. It starts monitoring for a stop-bit when all the bits in a command are received. If a stop-bit is received, it notifies RXC to read the command received from its register.

4.5 Interaction between Debugging Hardware and Debugging GUI

The debugging hardware (DH) and the debugging GUI (DGUI) exchanges commands and data through the RS-232 connection and is explained in previous sections. In this section, we will explain the transmission method and data transfered between the DH and the DGUI.

The DH is built-in hardware on the GPEAT system whereas DGUI is software running on host PC. DGUI sends commands to the DH to obtain the hardware identification, GPEAT system information and the GPEAT data. Table 4.6 summarizes commands recognized by the DH. The DH responds by sending a package back to the DGUI for further processing. There are three types of packages that DH sends back to DGUI which are the identification package, system information package and chromosome package. These three types of packages have various lengths in bytes. Each package is followed by an End-of-Package (EOP) byte-cluster to indicate to the DGUI that a transaction has ended. An EOP consists of three signature bytes along with number of bytes sent preceding the EOP. Figure 4.16 shows the structure of EOP.

Command (Single byte)	Description	Response
"i" (0x69)	Request for system identification	Identification Package
"s" (0x73)	Request for GPEAT system information	GPEAT System Information Package
"r" (0x72)	Request for chromosome information	Chromosome Package

Table 4.6 Commands recognized by Debugging Hardware

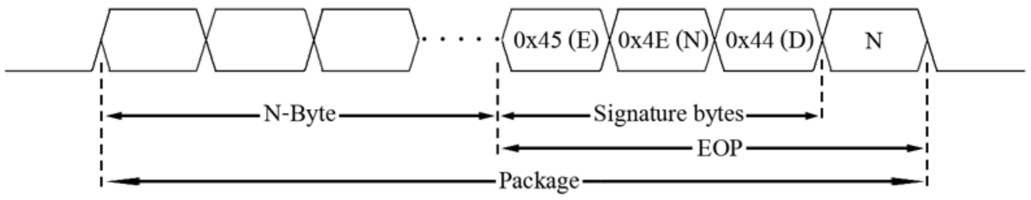


Figure 4.16 Structure of EOP

Figure 4.16 above shows the response from the DH through RS-232 connection. The package can be identification package, a GPEAT System information package or a chromosome package. Every type of packages will end with an EOP as shown above. The first N-Bytes of the package contain data or information sent back from the DH. The last four bytes make up the EOP byte-cluster. The first three bytes of the EOP are called the *signature bytes*. They are used to indicate to the DGUI that the package is ending at that particular point. The signature bytes consist of three bytes which are 0x45 (E), 0x4E (N) and 0x44 (D). The last byte of the EOP indicates the number of data or information bytes that have been transferred before the occurrence of EOP.

When the DGUI receives a package it reads and buffers the last byte of the package which is the last byte of the EOP. Then it checks if the signature bytes of EOP are valid (containing 0x45 [E], 0x4E [N] and 0x44 [D]). If the EOP is valid, it checks whether the amount of data in bytes arrived as specified in the last byte of EOP. In conclusion, if all conditions described are valid, it accepts the received package. If any of these conditions are not satisfied, it requests for resending of the package. Section 4.5.1, 4.5.2 and 4.5.3 will describe the structure of each type of package.

4.5.1 Identification Package

The identification package is sent back to host PC when DH received “i” (0x69) command from host PC. This type of package is normally sent back when DGUI initiates connections with the DH to check the validity of DH and verify the version of DH and GPEAT system. The structure of Identification Package is shown in Figure 4.17.

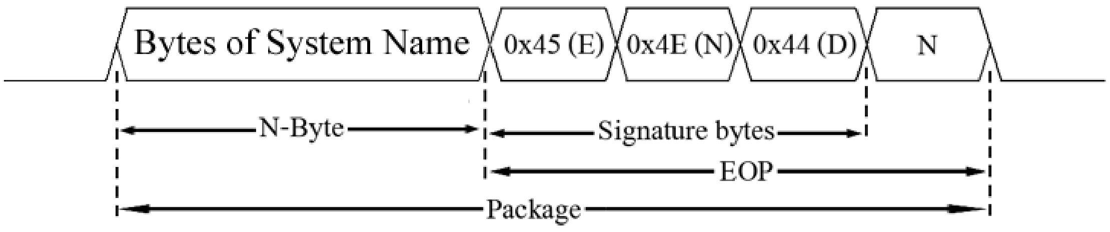


Figure 4.17 Identification Package

4.5.2 GPEAT System Information Package

The GPEAT system information package carries information on the GPEAT system implemented. This type of package will be sent back whenever DGUI request for system information by sending command “s” (0x73) to DH. The structure of GPEAT System Information Package is shown below in Figure 4.18.

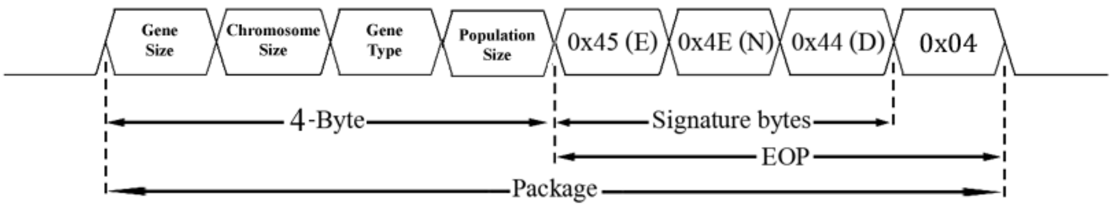


Figure 4.18 GPEAT System Information Package

4.5.3 Chromosome Package

The chromosome package is the most frequently requested package. Only one chromosome can be sent within a chromosome package. DGUI requests for chromosome packages by sending “r” (0x72) command to DH. The structure of chromosome package is shown in Figure 4.16.

CHAPTER 5

DEBUGGING GRAPHICAL USER INTERFACE

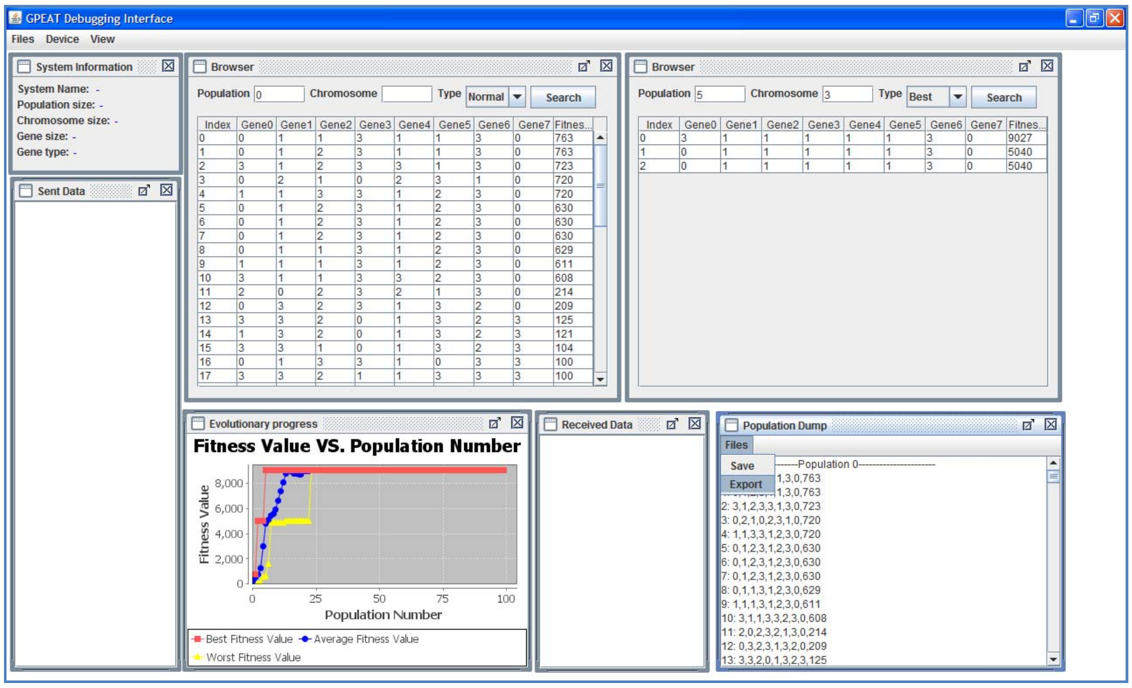
5.1 Introduction

The debugging graphical user interface (DGUI) provides an access for user to observe the progress of evolutionary process on the GPEAT system. Graphical and numeric representation of data obtained from GPEAT system is presented in the DGUI. DGUI performs as a storage system for all populations of chromosomes generated from GPEAT and can also be used as an analysis tool. It does not assist in any operations performed in GPEAT system but obtains data from the GPEAT system for debugging and analysis purposes. The DGUI interfaces with GPEAT system through debugging hardware described in chapter 4. In this chapter, the structure and features of DGUI is described.

5.2 Debugging Graphical User Interface (DGUI)

5.2.1 Workspace

The workspace is the area that contains the windows for displaying information obtained from the GPEAT system. The workspace window is shown in Figure 5.1.



System Name: -
Population size: -
Chromosome size: -
Gene size: -
Gene type: -

Population 0 Chromosome Type Normal

Index	Gene0	Gene1	Gene2	Gene3	Gene4	Gene5	Gene6	Gene7	Fitnes...
0	0	1	1	3	1	1	3	0	763
1	0	1	2	3	1	1	3	0	763
2	3	1	2	3	3	1	3	0	723
3	0	2	1	0	2	3	1	0	720
4	1	1	3	3	1	2	3	0	720
5	0	1	2	3	1	2	3	0	630
6	0	1	2	3	1	2	3	0	630
7	0	1	2	3	1	2	3	0	630
8	0	1	1	3	1	2	3	0	629
9	1	1	1	3	1	2	3	0	611
10	3	1	1	3	3	2	3	0	608
11	2	0	2	3	2	1	3	0	214
12	0	3	2	3	1	3	2	0	209
13	3	3	2	0	1	3	2	3	125
14	1	3	2	0	1	3	2	3	121
15	3	3	1	0	1	3	2	3	104
16	0	1	3	3	1	0	3	3	100
17	3	3	2	1	1	3	3	3	100

Population 5 Chromosome 3 Type Best

Index	Gene0	Gene1	Gene2	Gene3	Gene4	Gene5	Gene6	Gene7	Fitnes...
0	3	1	1	1	1	1	3	0	9027
1	0	1	1	1	1	1	3	0	5040
2	0	1	1	1	1	1	3	0	5040

```
-------Population 0----------------------
1,3,0,763
1,3,0,763
2: 3,1,2,3,3,1,3,0,723
3: 0,2,1,0,2,3,1,0,720
4: 1,1,3,3,1,2,3,0,720
5: 0,1,2,3,1,2,3,0,630
6: 0,1,2,3,1,2,3,0,630
7: 0,1,2,3,1,2,3,0,630
8: 0,1,1,3,1,2,3,0,629
9: 1,1,1,3,1,2,3,0,611
10: 3,1,1,3,3,2,3,0,608
11: 2,0,2,3,2,1,3,0,214
12: 0,3,2,3,1,3,2,0,209
13: 3,3,2,0,1,3,2,3,125
```

Figure 5.1 DGUI workspace

The windows contained in workspace will be visible only when user chooses to view them by accessing the *View* menu on the menu bar. On the menu bar, the user can open or save previously obtained data to/from a file by accessing *Files* menu. The user can also connect/disconnect to GPEAT system, start/stop evolutionary process on GPEAT and configure RS-232 connection by accessing *Device* menu. To view obtained data that has been sent from GPEAT, the user would select from the *View* menu.

5.2.1.1 Loading debugging data

The user can open debugging data that has previously been received from the GPEAT system without connecting to the system. Debugging data is stored in a binary format with file extension of .gdi. A .gdi file stores all chromosomes in all populations along with their fitness values. A .gdi file is opened by clicking *Files* → *Open*.

5.2.1.2 Saving debugging data

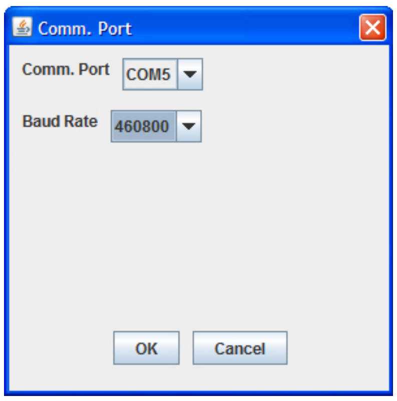
After receiving data from the GPEAT system, the user can save this data in a .gdi file for later analysis. Received data can be saved by clicking *File* → *Save*.

5.2.2 Working with GPEAT hardware

The DGUI connects with GPEAT system through an RS-232 connection. In order to connect to the built-in debugging hardware (described in chapter 4) with DGUI, the user has to enable the built-in debugging hardware in the parameter entry GUI (described in chapter 3). Otherwise, the DGUI will still connect to the GPEAT system but it will not be able to obtain data. The following section 5.2.2.1 and 5.2.2.2 assume that the built-in debugging hardware is enabled and show the steps for connecting and obtaining data from the GPEAT system.

5.2.2.1 Connecting to GPEAT hardware

Before connecting to the GPEAT system, the user is required to configure an RS-232 connection which will be used as the channel for communication. Figure 5.2 shows the setup window for configuring RS-232 connection. This window can be accessed by clicking *Device* → *Setup*.



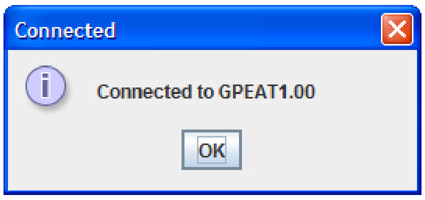
Comm. Port COM5 (1)

Baud Rate 460800 (2)

Figure 5.2 RS-232 Setup

There are two parameters that have to be specified in the setup window which are communication port (1) and baud rate (2). The communication port drop-down list (1) shows all available physical RS-232 ports on the computer that can be used to connect to the GPEAT system. The user has to select only one port that is connected. The baud rate determines the speed of the data transfer. If the baud rate selected in this setup window does not matches with the speed set in the system, no communication will be established. In this literature, we chose communication port as COM5 and baud rate is 460800 bit-per-second.

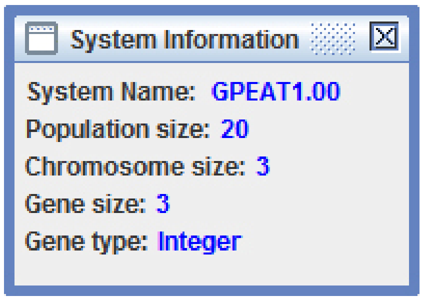
After configuring the RS-232 connection, the next step is to connect to the hardware. This step verifies the hardware signature and version and obtains the GPEAT parameters previously configured through the parameter entry GUI. To connect to the hardware, user selects *Device* → *Connect*. Figure 5.3 shows a dialog showing hardware signature and version.



Connected to GPEAT1.00

Figure 5.3 A dialog showing hardware signature and version

After the hardware is recognized by the DGUI, the DGUI requests the GPEAT parameters by sending a request-for-information-package command (described in chapter 4) to GPEAT. The window shown in Figure 5.4 shows the GPEAT parameters obtained from the hardware. The window can be toggled between show and hide by selecting *View* → *System Information* from menu bar.



System Name: GPEAT1.00
Population size: 20
Chromosome size: 3
Gene size: 3
Gene type: Integer

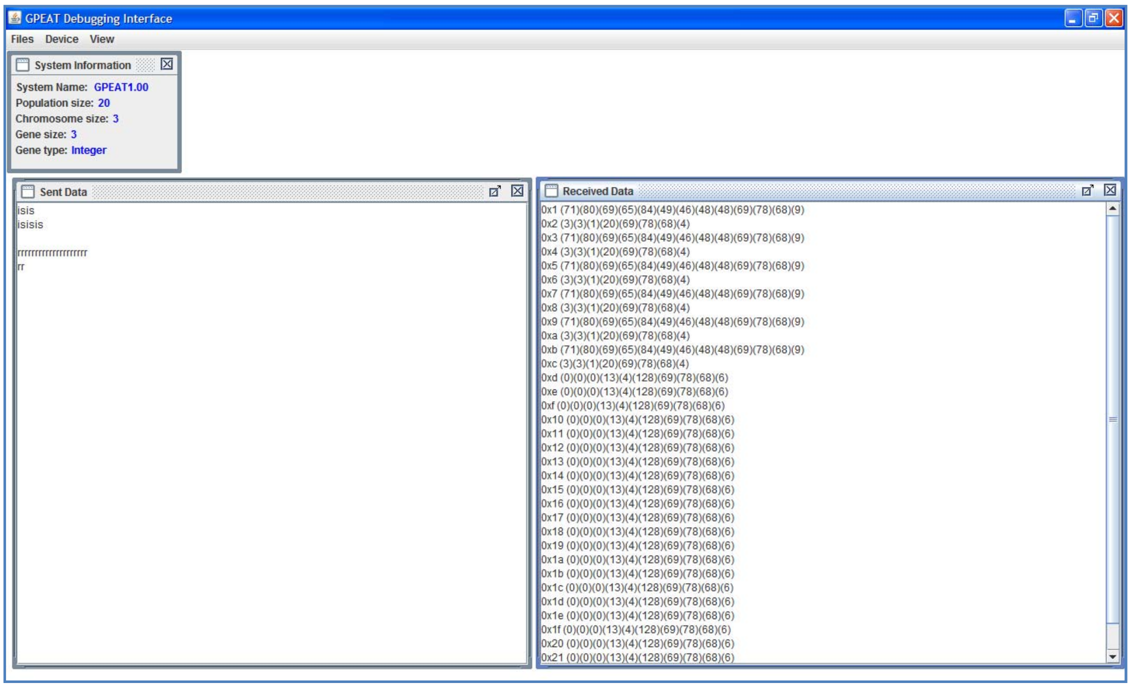
Figure 5.4 A window showing GPEAT parameters

5.2.2.2 Obtaining data from hardware

When the connection has been established, DGUI sends a request to GPEAT to send the chromosome information and then stores that information in memory. The DGUI provides two ways to request chromosomes information and they are continuous requests and manual requests.

In the continuous request mode, the DGUI continuously sends request-for-chromosome-package command to the hardware until the evolutionary process stops. Each chromosome will be processed and stored in memory which is accessible to the user. Notice that when the debugging hardware is enabled (by parameter entry through the GUI), the evolutionary process cannot be started without interaction by the user through the DGUI interface. It is necessary to start the process through DGUI and then the DGUI will grab all chromosomes and store them. The user can start the process by selecting *Device* → *Start* after connecting to the hardware.

In the manual request mode, the DGUI provides an interfacing window for sending commands to the hardware. The user can send the commands defined in chapter 4 by typing commands in this window. If the request-for-chromosome-package command is initiate by the user, the chromosome packages received will also be processed and stored in a memory that is accessible to the user and can be used for debugging and analysis purposes. Figure 5.5 shows the interfacing windows contained in DGUI workspace.



System Name: GPEAT1.00
Population size: 20
Chromosome size: 3
Gene size: 3
Gene type: Integer

```
isis
isisis

rrrrrrrrrrrrrrrrrrr
rr
```

```
0x1 (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0x2 (3)(3)(1)(20)(69)(78)(68)(4)
0x3 (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0x4 (3)(3)(1)(20)(69)(78)(68)(4)
0x5 (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0x6 (3)(3)(1)(20)(69)(78)(68)(4)
0x7 (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0x8 (3)(3)(1)(20)(69)(78)(68)(4)
0x9 (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0xa (3)(3)(1)(20)(69)(78)(68)(4)
0xb (71)(80)(69)(65)(84)(49)(46)(48)(48)(69)(78)(68)(9)
0xc (3)(3)(1)(20)(69)(78)(68)(4)
0xd (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0xe (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0xf (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x10 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x11 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x12 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x13 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x14 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x15 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x16 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x17 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x18 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x19 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1a (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1b (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1c (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1d (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1e (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x1f (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x20 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
0x21 (0)(0)(0)(13)(4)(128)(69)(78)(68)(6)
```

Figure 5.5 Interfacing windows in a workspace

(1) and (2) in Figure 5.5 are "Sent Data" and "Received Data" windows. The user can manually type a command in (1) and see the received bytes in (2). Each character in (1) is recognized as a command and each value in parenthesis in (2) represents values of received bytes. Each line in (2) also represents an each individual package received from the hardware.

5.2.3 Evolutionary Progress Graph

In EC applications, evolutionary progress observation is one of the important performance measurements. Fitness-value tracking is the common method of evolutionary progress observation for many EC applications. The best, average and worst fitness values are plotted against their generation number which shows time to convergence for the algorithm. The best fitness values are picked from the chromosomes to generate the best fitness curve, and worst values are picked from ones that have lowest fitness values to generate the worst fitness curve. The average fitness values used to create the mean of the fitness value curve. The evolutionary progress graph can be used to check if the algorithm has converged. A graph which shows the progress for evolutionary process is shown in Figure 5.6.

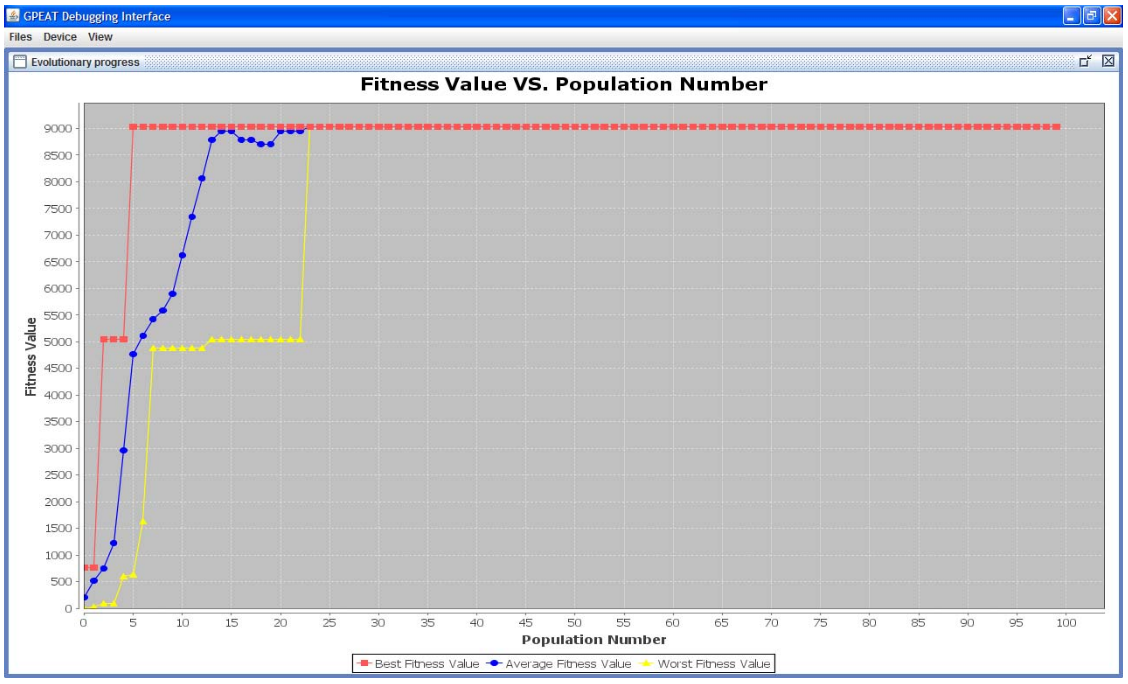
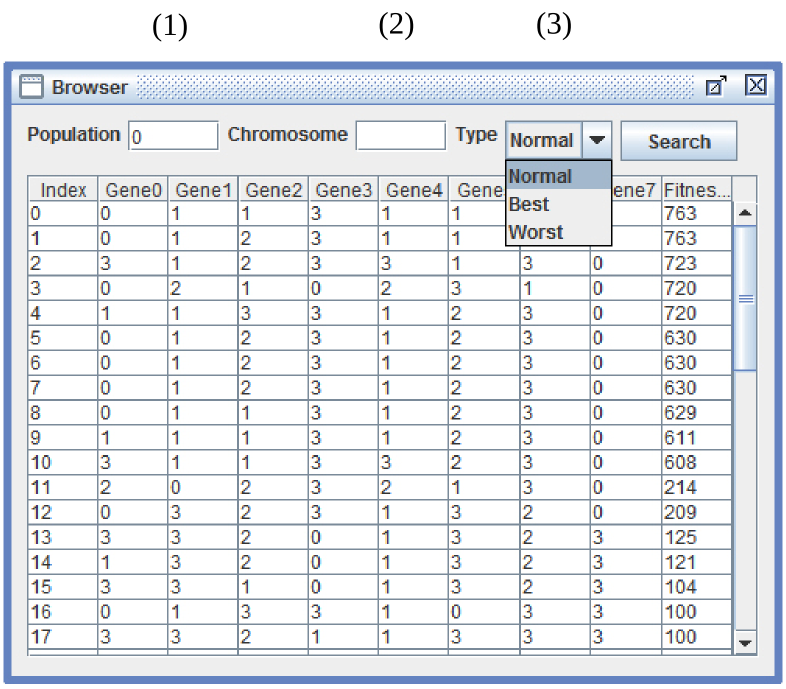


Figure 5.6 Evolutionary Progress Graph

5.2.4 Browser

A Browser is a window which is provided for browsing chromosomes from your choice of generation. The user can have as many browser windows open as he needs. A browser window can be added to the workspace by selecting *View* → *New Browser*. Figure 5.7 shows a browser window.



Population 0 Chromosome Type Normal

Index	Gene0	Gene1	Gene2	Gene3	Gene4	Gene		ene7	Fitnes...
0	0	1	1	3	1	1			763
1	0	1	2	3	1	1			763
2	3	1	2	3	3	1	3	0	723
3	0	2	1	0	2	3	1	0	720
4	1	1	3	3	1	2	3	0	720
5	0	1	2	3	1	2	3	0	630
6	0	1	2	3	1	2	3	0	630
7	0	1	2	3	1	2	3	0	630
8	0	1	1	3	1	2	3	0	629
9	1	1	1	3	1	2	3	0	611
10	3	1	1	3	3	2	3	0	608
11	2	0	2	3	2	1	3	0	214
12	0	3	2	3	1	3	2	0	209
13	3	3	2	0	1	3	2	3	125
14	1	3	2	0	1	3	2	3	121
15	3	3	1	0	1	3	2	3	104
16	0	1	3	3	1	0	3	3	100
17	3	3	2	1	1	3	3	3	100

Figure 5.7 A Browser window

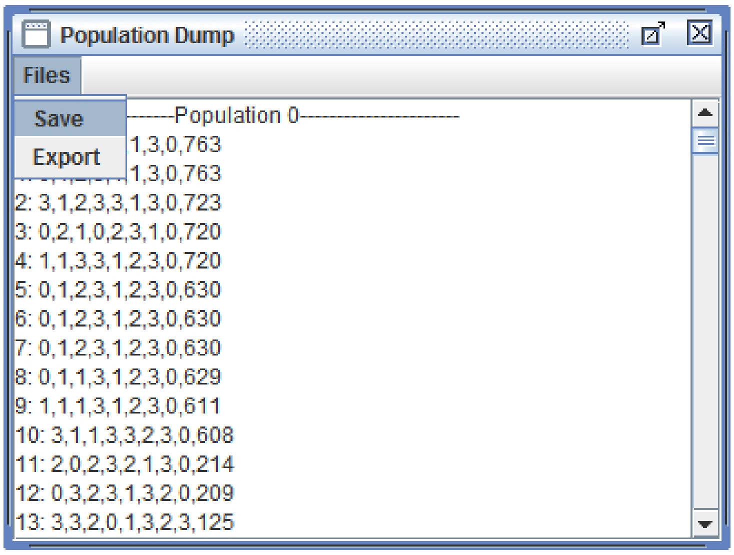
Three mode of browsing are provided and they are the normal mode, the best mode and the worst mode. Two numerical text inputs (1), (2) and one drop-down list (3) are used for browsing chromosomes and selecting browsing mode. Numerical text inputs (1) and (2) are used to specify the population number and the index-of-chromosome/number-of-chromosome respectively. Drop-down list (3) is used for selecting browsing mode (Normal/Best/Worst).

Normal mode shows all the chromosomes contained in a specified population, if no index of chromosome is specified. One chromosome will be shown, if there is an index of a chromosome specified. The user specifies the population number that will be accessed in (1) and the index of chromosome in (2). (2) can be left blank if user wishes to view all chromosomes in that particular population.

The best mode browses for a specified number of the best chromosomes in a specified population, while the worst mode browses for a specified number of worst chromosomes in a specified population. The user specifies the population number (generation) that will be accessed in (1) and a number of best/worst chromosomes in (2).

5.2.5 Populations Dump

The population dump window shows all chromosomes in all populations. Chromosomes are shown in text form with indexes, contents (genes) and fitness values. Figure 5.8 shows population dump window. Text generated can either be saved to a text file or exported in a spreadsheet format for further analysis using a spreadsheet application like Microsoft Excel. To save or export text, the user selects *Files* → *Save* or *Files* → *Export* respectively from the file menu in the population dump window.



```
-------Population 0---------------------
1,3,0,763
1,3,0,763
2: 3,1,2,3,3,1,3,0,723
3: 0,2,1,0,2,3,1,0,720
4: 1,1,3,3,1,2,3,0,720
5: 0,1,2,3,1,2,3,0,630
6: 0,1,2,3,1,2,3,0,630
7: 0,1,2,3,1,2,3,0,630
8: 0,1,1,3,1,2,3,0,629
9: 1,1,1,3,1,2,3,0,611
10: 3,1,1,3,3,2,3,0,608
11: 2,0,2,3,2,1,3,0,214
12: 0,3,2,3,1,3,2,0,209
13: 3,3,2,0,1,3,2,3,125
```

Figure 5.8 Population Dump window

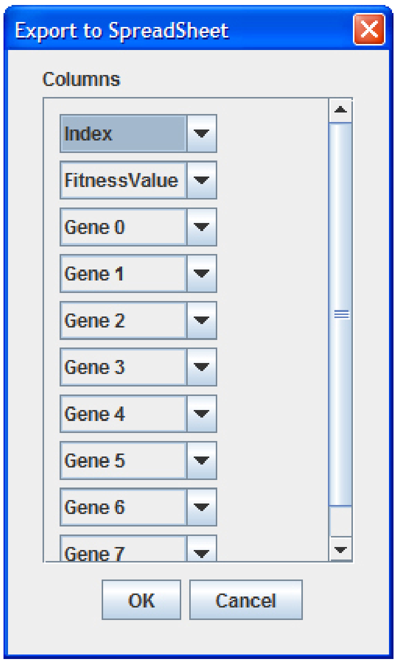


Figure 5.9 Export to SpreadSheet dialog

By selecting File → Export in the population dump window, a dump of all the chromosomes to a file in spreadsheet format will be carried out. The window which is shown in Figure 5.9 shows headers for columns for the spreadsheet dump. The user can customize the header order and inclusion and choose whether headers will be visible or hidden in the spreadsheet. After clicking OK, the user will be prompted for a filename for the file to be saved to. This will store all chromosomes in spreadsheet format to a specified text file.

CHAPTER 6

SIMULATION AND DATA

6.1 Hardware simulation

In this chapter the system is simulated and the debugger abilities are demonstrated.

In the hardware simulation, the debugging hardware is broken down into the following modules: The DC (debugger controller), TXC (transmitting controller), TXI (transmitting interface), RXC (receiving controller), and RXI (receiving interface). The modules are tested separately to verify their individual functionalities and analyze signal integrity. Each module is tested with a testbench that is specially design for that particular module. A testbench is an entity written in VHDL which is not synthesizable but provides the essential signals for a module in order to function normally in a simulation. We use ModelSim XE-III (MXE-III) to obtain the simulation results. Once the VHDL files for debugging the hardware are compiled together with the testbenches then, simulations can be started and compatibility among the modules can be tested.

6.1.1 The Debugger Controller

When the Debugger Controller (DC) receives one of the following commands request-for-identification-package (0x69), request-for-information-package (0x73) or request-for-chromosome-package (0x72), it will carry out the actions described earlier chapter 4. In the simulation, the DC is tested with all of the commands with the simulation setup shown in Table 6.1.

Simulation Setup

Parameters	**Value**
Clock speed	50 MHz (50% Duty cycle)
Baud rate	462962.963 bit-per-second
System Identification	"GPEAT1.00"
Population size	20
Chromosome size	5
Gene size	10
Gene type	Integer
Test Chromosome	Gene0: 0x0BC (0010111100B, 10-bit) Gene1: 0x303 (1100000011B, 10-bit) Gene2: 0x07F (0001111111B, 10-bit) Gene3: 0x3FE (1111111110B, 10-bit) Gene4: 0x3F7 (1111110111B, 10-bit) Bytes: - 0x2F (47D) - 0x30 (48D) - 0x31 (49D) - 0xFF (255D)

	- 0xFE (254D) - 0xFD (253D) - 0xC0 (192D)
Test Fitness Value	0x DE7B7EFF (3732635391D) Bytes: - 0xDE (222D) - 0x7B (123D) - 0x7E(126D) - 0xFF (255D)

Table 6.1 Simulation Setup for Debugger Controller (DC)

As noted in Table 6.1, the test chromosome has five 10-bit integer numbers for five genes. All bit strings representing values will be concatenated to form a 50-bit bit-string in order to be processed by the DC. Because that 50-bit string cannot be exactly divided into equal number of bytes, the last two bits have six zeros concatenated in order to form one whole byte. Once the zeros are added to the last byte the string will have 56-bits and can be chopped into 7 bytes. These 7 bytes contain same information as the chromosome does. The chromosome will be reconstructed with the Debugging GUI (DGUI) and the six unused zeros will be ignored by the DGUI. The process of concatenation will be internally done by the DC. The GPEAT core is only responsible for feeding unprocessed chromosome to the DC. We only want to clarify the process of sending chromosome back in this chapter.

Simulation Result

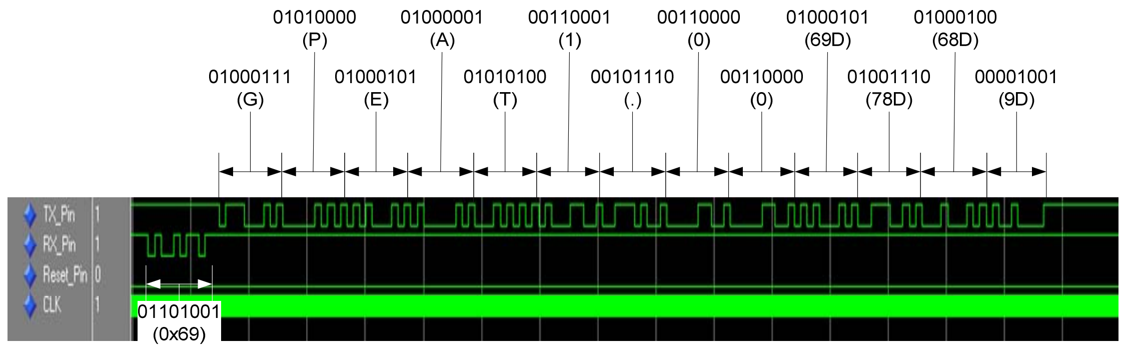


Figure 6.1 Timing diagram for response of 0x69 command

Figure 6.1 shows the timing diagram for RS-232 connection which is the simulation result of testing the DC with the 0x69 command (request-for-identification package). The command is fed through the *RX Pin* as shown and the *TX Pin* shows the response of the DC. The identification string is ‘GPEAT1.00’ as shown in Table 6.1. This string is sent out followed by EOP as shown in Figure 6.1.

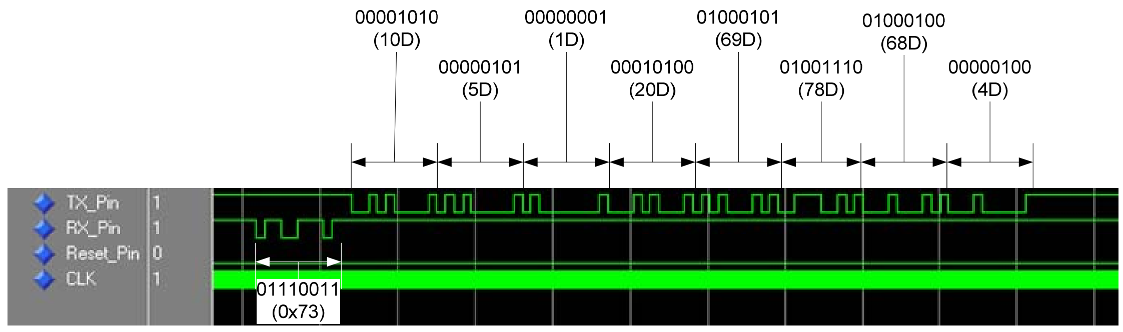


Figure 6.2 Timing diagram for response of 0x73 command

Figure 6.2 shows the timing diagram for the simulated result of sending the 0x73 command (request-for-information-package) to the DC. Gene size is sent out first followed by chromosome size, gene type and population size. In this particular case, the first byte contains 10D which is the gene size, 5D which is the chromosome size for the second byte, 1D for the third byte indicating integer-typed gene is used and 20D for the fourth gene specifying the population size. The EOP also appears here as the last byte-cluster.

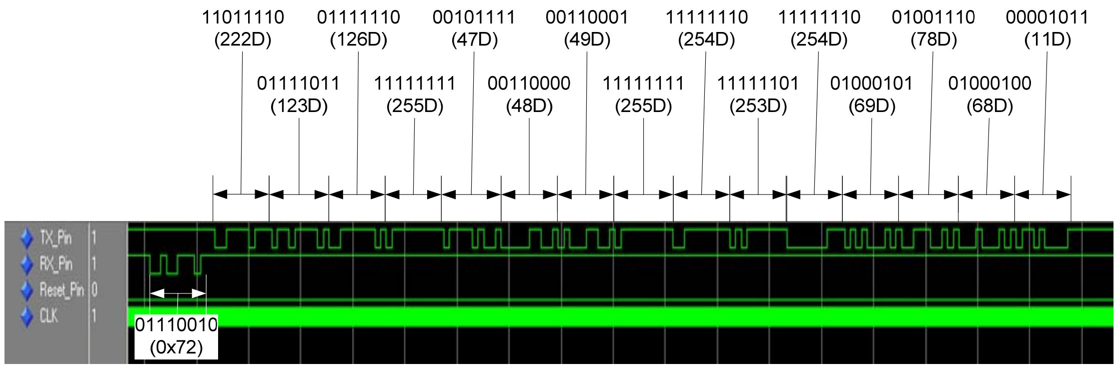


Figure 6.3 Timing diagram for response of 0x72 command

The test chromosome is sent back in this simulation result. A 0x72 command is sent to the DC. In response, the DC sends the chromosome followed by an EOP back. In Figure 6.3, 11 individual bytes are sent back sequentially. The first four bytes contain 32-bit integer number for fitness value. Following 7 bytes contains the chopped chromosome shown earlier in Table 6.1.

6.1.2 The TX Controller (TXC)

The simulation shown in this section shows the internal interactions between the TXC and the DC. The testbench for the TXC does not act exactly the same as the DC does but it provides adequate control signals to verify the functionality of the TXC. In this simulation, TX FIFO, TXC and TXI are also being verified for their functionality since the TX FIFO and TXI are considered to be built-in modules in the TXC. Table 6.2

shows the simulation setup. FIFO size is limited to four bytes and four test-vectors are provided.

Simulation Setup

Parameters	**Value**
Clock speed	50 MHz (50% Duty cycle)
Baud rate	462962.963 bit-per-second
FIFO size	4 bytes
Data bytes (Test-Vector)	1: 0x55 (01010101B, 85D) 2: 0xF0 (11110000B, 240D) 3: 0x0F (00001111B, 15D) 4: 0xCC (11001100B, 204D)

Table 6.2 Simulation Setup for TX Controller (TXC)

In this simulation, all 4 test-vectors will be clocked into the FIFO while the TXC is operating in mode-0. Then, the mode of TXC will be changed to mode-1 in which the test-vectors will be transferred serially through *TX Pin*.

Simulation Result

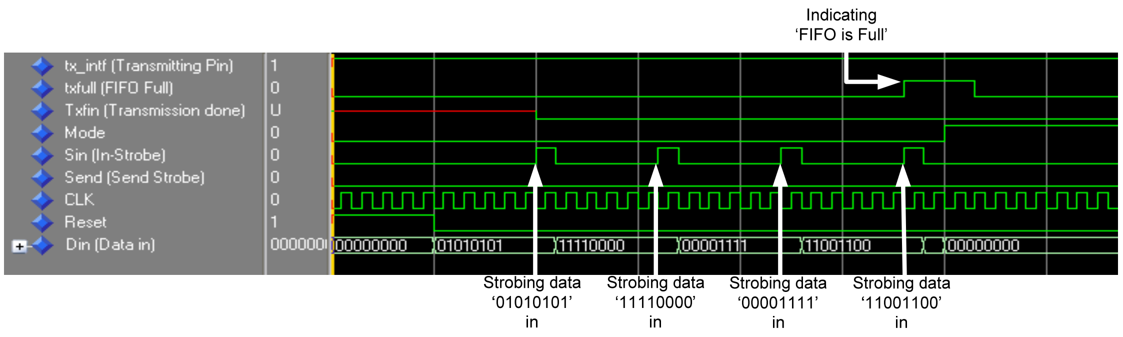


Figure 6.4 Timing diagram for TXC mode-0

Figure 6.4 shows the process of clocking four test-vectors in the TX FIFO through the TXC in mode-0. The four arrows on the bottom of Figure 6.4 indicate the rising-edges of the *In-Strobe* signal which is used to tell the TXC to grab the data (test-vector) being presented at *Din* (Data-in) pins. The order for clocking test-vectors in is 0x55, 0xF0, 0x0F and 0xCC respectively. When all test-vectors have been clocked in, the FIFO is full and TXC notifies the DC that the FIFO is full by sending a *txfull* signal. The arrows on the top of Figure 6.4 shows when *txfull* is high indicating that no more data (test-vectors) can be accepted.

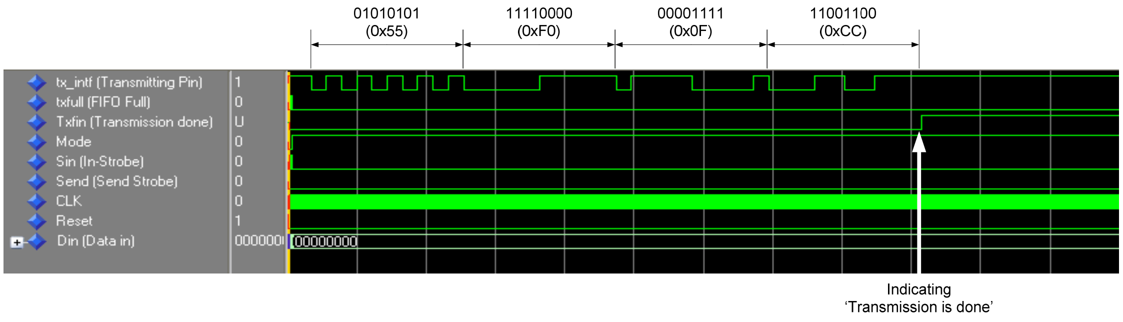


Figure 6.5 Timing diagram for TXC mode-1

After all test-vectors are clocked into the the FIFO of TXC, the testbench changes the TXC mode to mode-1. In mode-1, TXC automatically checks the availability of the data stored in the FIFO and sends data bytes out. Figure 6.5 shows that test-vectors are sent out through *tx_intf* (Transmitting Pin) in the same order as the order of clocking that test-vectors came in. In Figure 6.5, the first byte, which is 0x55, is sent out followed by 0xF0, 0x0F and then 0xCC.

6.1.3 The TX Interface (TXI)

The TXI is the module that transmits data serially. The testbench provides test-vectors to the TXI in parallel (8 terminals for 1 byte), and the TXI transmit this data serially. The TXI has two sets of Data-in terminals (2 sets of 8-terminal inputs), Din0 and Din1. After clocking data through one input, the next data can be clocked in through the other after a *rdy* (Ready) signal goes low and then back again to high. The test-vectors and simulation setup are shown in Table 6.3.

Simulation Setup

Parameters	**Value**
Clock speed	50 MHz (50% Duty cycle)
Baud rate	462962.963 bit-per-second
Data bytes (Test-Vector)	1: 0xAA (10101010B, 170D) 2: 0x55 (01010101B, 85D) 3: 0xF0 (11110000B, 240D) 4: 0x0F (00001111B, 15D)

Table 6.3 Simulation Setup for TX Interface (TXI)

Simulation Result

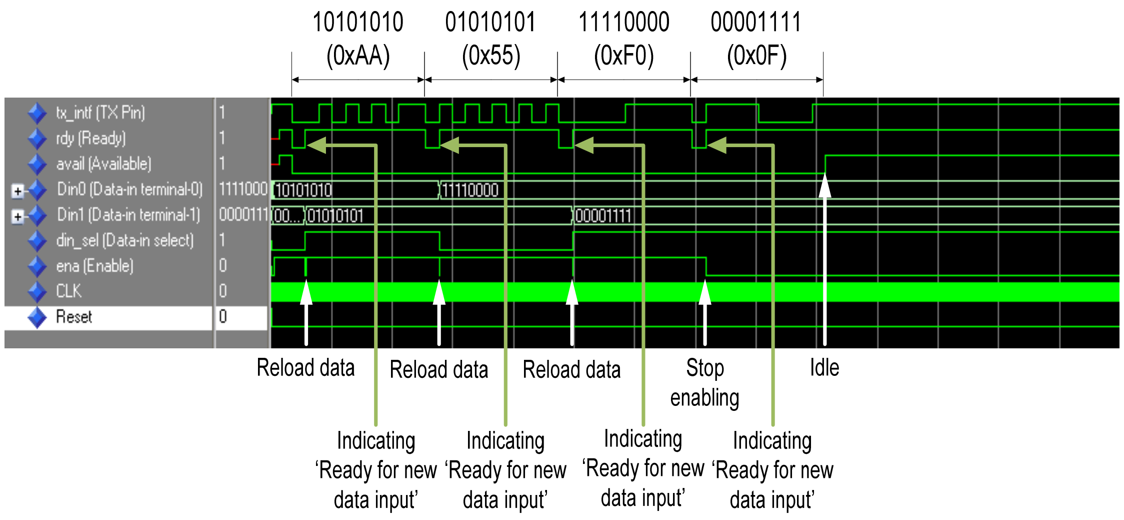


Figure 6.6 Timing diagram for TXI

As shown in Figure 6.6, *Din0* and *Din1* data inputs are used to clock the test-vectors in. *Din_sel* indicates the active data input and is toggled between test-vectors. The first test-vector is fed through *Din0*, the second through *Din1*, the third through *Din0* and the fourth through *Din1*. When the first bit of first vector (0xAA) is sent out, RXI pulls *rdy* low and then back high again when the second bit of the first test-vector is being sent. This indicates that RXI is ready to grab new data from the other data input. The second test-vector is then fed through *Din1*.

6.1.4 The RX Controller (RXC)

The simulation for the RXC is as same as the TXC. It shows the internal interactions between the DC and the RXC but the DC does not act exactly the same as it did in the TXC simulation. In this test, the integration of RXC, RX FIFO and RXI is verified. Table 6.4 shows the simulation setup and test-vectors used to verify these blocks' functionality.

Simulation Setup

Parameters	**Value**
Clock speed	50 MHz (50% Duty cycle)
Baud rate	462962.963 bit-per-second
FIFO size	4 bytes
Data bytes (Test-Vector)	1: 0xAA (10101010B, 170D) 2: 0x55 (01010101B, 85D) 3: 0xF0 (11110000B, 240D) 4: 0x0F (00001111B, 15D)

Table 6.4 Simulation Setup for RX Controller (RXC)

Simulation Result

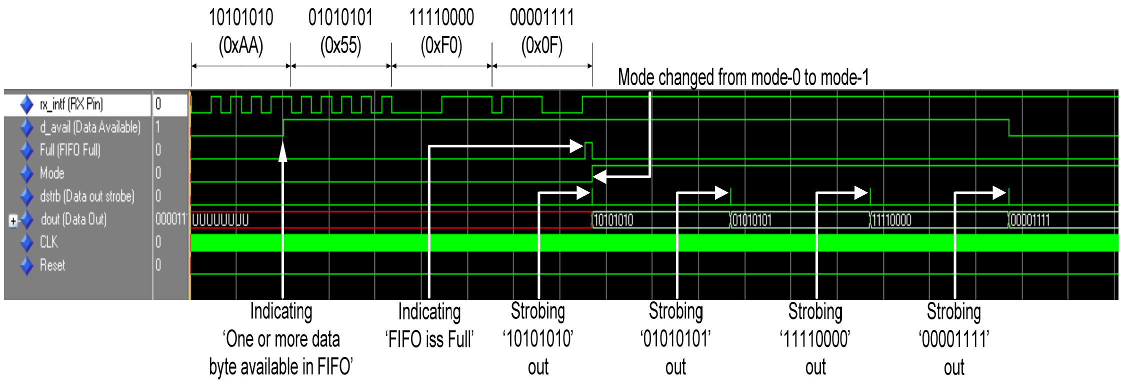


Figure 6.7 Timing diagram for RXC

Because the RXC controls the receiving process that receives data serially from RS-232 connection, the test-vectors are fed through the *rx_intf* (RX Pin). When the RXC receives data through *rx_intf*, it stores a complete transmission of received data in the FIFO and waits for the data to be clocked out. In Figure 6.7, all four test-vectors are fed serially through *rx_intf* as the RXC stores the completely received transmission of data simultaneously in mode-0. When all four test-vectors have been received, the RXC raises the *Full* signal to high indicating that the FIFO is full. To clock the data out, *Mode* is changed to mode-1 and *dstrb* (Data-out strobe) is used to signify to the RXC when to present the data at *Dout* (Data-out terminal). New data will be presented when RXC receives rising-edge of *dstrb*.

6.1.5 The RX Interface (RXI)

In the simulation for RXI, the testbench provides test-vectors to RXI serially to verify the correct performance of the receiving mechanism of RXI. RXI has only one register to store received data; so, the received data has to be clocked out immediately in order to receive new data. After the testbench provides a complete test-vector to RXI, it immediately clock the data out from RXI register before feeding a new test-vector in. The simulation setup and test-vectors are shown in Table 6.5.

Simulation Setup

Parameters	**Value**
Clock speed	50 MHz (50% Duty cycle)
Baud rate	462962.963 bit-per-second

Data bytes (Test-Vector)	1: 0xAA (10101010B, 170D) 2: 0x55 (01010101B, 85D) 3: 0xF0 (11110000B, 240D) 4: 0x0F (00001111B, 15D)

Table 6.5 Simulation Setup for RX Interface (RXI)

Simulation Result

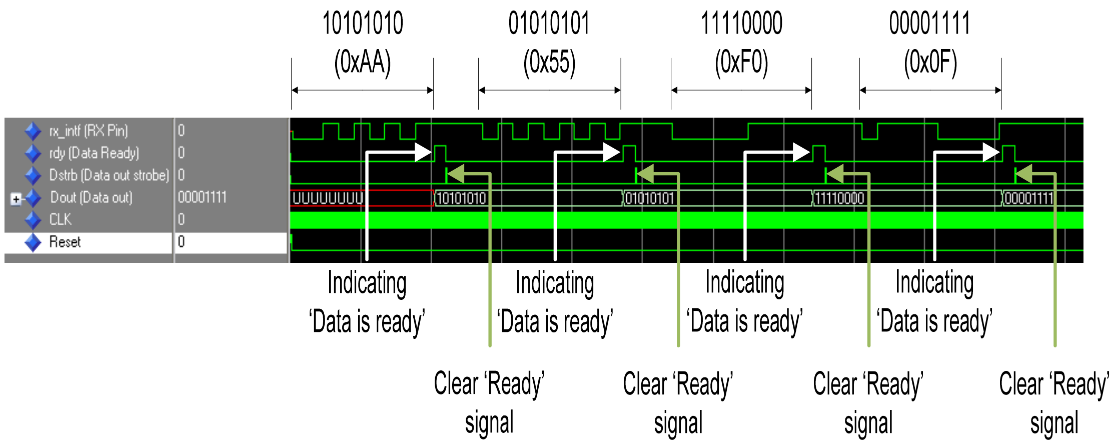


Figure 6.8 Timing diagram for RXI

As shown in Figure 6.8, test-vectors are fed in through *rx_intf* (RX Pin). When a complete test-vector (data) is received, RXI raises the *rdy* (Data Ready) signal high, indicating that new data has been received. After the testbench has seen the *rdy* signal, it clocks the data out by providing a strobe, Dstrb (Data-out strobe), to RXI. RXI will present the new data at the Dout (Data-out) terminals after seeing a rising-edge on the *Dstrb* signal.

6.2 Hardware Usage

In this section, we analyze the hardware usage for the debugging hardware to see the resource consumption of the debugging hardware. Each module is observed for FPGA resource utilization. In this analysis, the debugging hardware is attached to a dummy GPEAT core which acts the same as a GPEAT core would except that the same chromosome will be sent back to host PC for each parameter setting. The dummy GPEAT core and the debugging hardware are synthesized according to the GPEAT parameters. We fixed the gene size to 4 bits and gene type to binary. Chromosome size is varied from 2 to 14 which yields a range of bits for one chromosome from 8 to 56.

	4x2	**4x3**	**4x4**	**4x5**	**4x6**	**4x7**	**4x8**	**4x9**	**4x10**	**4x11**	**4x12**	**4x13**	**4x14**
#Memory	3	3	3	3	3	3	3	3	3	3	3	3	3
Dual-port distributed RAM	14x8x2	14x8x2	15x8x2	15x8x2	16x8x2	16x8x2	17x8x2	17x8x2	18x8x2	18x8x2	19x8x2	19x8x2	20x8x2
ROMS	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1	10x8x1
#Adder/Subtractors	4	4	4	4	4	4	4	4	4	4	4	4	4
4-bit adder	2	2	2	2	2	2	2	2	2	2	2	2	2
8-bit adder	1	1	1	1	1	1	1	1	1	1	1	1	1
8-bit subtractor	1	1	1	1	1	1	1	1	1	1	1	1	1
#Counters	11	11	11	11	11	11	11	11	11	11	11	11	11
16-bit up counter	4	4	4	4	4	4	4	4	4	4	4	4	4
4-bit up counter	5	5	5	5	5	5	1	1	1	1	1	1	1
5-bit up counter	0	0	0	0	0	0	4	4	4	4	4	4	4
4-bit updown counter	2	2	2	2	0	0	0	0	0	0	0	0	0
5-bit updown counter	0	0	0	0	2	2	2	2	2	2	2	2	2
#Register	153	153	153	153	153	153	153	153	153	153	153	153	153

Flip-Flops	153	153	153	153	153	153	153	153	153	153	153	153	153
#Latches	9	10	10	11	11	12	12	13	13	14	14	15	15
1-bit latch	2	2	2	2	2	2	2	2	2	2	2	2	2
8-bit latch	7	8	8	9	9	10	10	11	11	12	12	13	13
#Comparators	7	7	7	7	7	7	7	7	7	7	7	7	7
16-bit comparator less	4	4	4	4	4	4	4	4	4	4	4	4	4
4-bit comparator gratequal	1	1	1	1	1	1	1	1	1	1	1	1	1
4-bit comparator less	2	2	2	2	2	2	2	2	2	2	2	2	2
#Multiplexers	1	1	1	1	1	1	1	1	1	1	1	1	1
8-bit 256-to-1 multiplexer	1	1	1	1	1	1	1	1	1	1	1	1	1
#Xors	2	2	2	2	2	2	2	2	2	2	2	2	2
1-bit xor2	2	2	2	2	2	2	2	2	2	2	2	2	2

Table 6.6 Basic Macros recognized

We use ‘Xilinx XST’ as the HDL synthesis tool. In the HDL synthesis process Xilinx XST tries to recognize basic macros, e.g. flip-flops, counters multiplexer and, in the overall design. After performing the synthesis process, Xilinx XST reports resource use and that resource use for this system is shown in Table 6.6. The headers indicate the gene size and chromosome size specified in parameter settings: For example, “4x2” means gene size of 4 and chromosome size of 2. The first column of each row shows the type of the macros recognized and the data in the table shows amount of a particular type of macro used.

From Table 6.6, we observe that as we increase the chromosome size the hardware resources used increases. The resources that increase notably are latches, counters and RAM. In our design of debugging hardware, we know that the number of latches used increases because as the chromosome size increases, the number of genes concatenated together is higher resulting in the increasing of the number of total bits—and, because, during processing all chromosomes are held locally in flip-flops rather than in RAM, the number of latches required to hold data increases. The number of latches used increases when the chromosome size increases to an odd number. This occurs because four additional bits are added into the system for each increase in the number of genes but the information is stored as bytes. So, even through only four new bits are needed, an 8-bit space is used. For an odd number of genes, the extra bits are filled with zeros. The four additional bits have to be filled because the RS-232 protocol transmits bytes. Filling extra bits with zeros was explained in section 6.1.1.

We found that as the chromosome size increased from 2 to 14 genes, the RAM also increased from 14 to 20 words. The RAM is also used as two separate macros. I found that the recognized RAM modules are not used as a memory unit of the debugging hardware because the operation of memory unit is to latch data which can be achieve by using D-flip-flops and multiplexer as shown in Figure 4.10. Another reason is that the word size of RAM modules is not 256. The word size of 256 words is fixed for the memory unit. The other units that can be using RAM are RX-FIFO and TX-FIFO because these units are also implemented using memory macros and the size can be varied according to our HDL design. Thus, we strongly believe that the dual-port RAM macros are used for implementing FIFO modules.

The result in Table 6.6 suggest us that the overall resource consumption does not depend much on the GPEAT core except for the memory modules (debugging hardware's memory unit and the FIFOs). To conclude this, we also show the mapping statistics for FPGA in Figure 6.9.

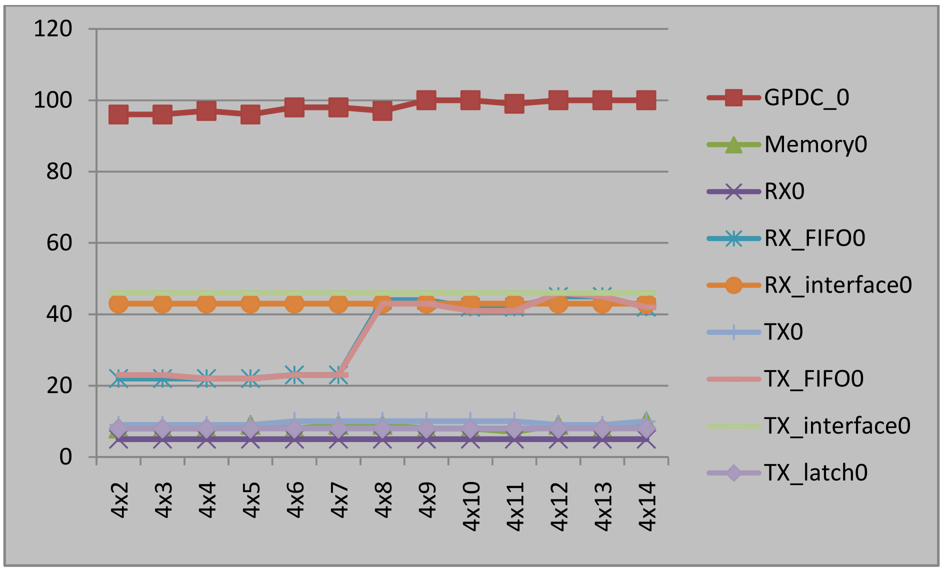


Figure 6.9 FPGA slices used for debugging hardware implementation

A slice is one of the basic units used to describe an FPGA. There are four slices in a single configurable logic block (CLB). A CLS includes resource to implement two four-input or one five-input logic block plus provides two flip-flops. We used a Spartan 3A chip which has 5968 total slices. We can see from Figure 6.9 that the numbers of slices used for FIFOs varied as the chromosome length increases from 7 to 8. The number of slices used by other components stay relatively constant. Table 6.7 shows the design summary for debugging hardware which is attached to the dummy GPEAT core (Chromosome size: 14, Gene size: 4).

Logic Utilization:	**Used**	**Available**	**Utilization**
Total Number Slice Registers:	267	11,776	2%
Number used as Flip Flops:	249		
Number used as Latches:	18		
Number of 4 input LUTs:	527	11,776	4%
Logic Distribution:			
Number of occupied Slices:	307	5,888	5%
Number of Slices containing only related logic:	307	307	100%
Number of Slices containing unrelated logic:	0	307	0%
Total Number of 4 input LUTs:	529	11,776	4%
Number used as logic:	463		
Number used as a route-thru:	2		
Number used for Dual Port RAMs:	64		

Number of bonded IOBs:	4	372	1%
Number of BUFGMUXs:	4	24	16%

Table 6.7 Design summary for debugging hardware

The design summary is for the implementation of the debugging hardware using the dummy GPEAT core on a Xilinx Spartan-3A FPGA. The total look-up-tables (LUTs) and Registers used in slices are 529 and 267 respectively which are 4% and 2% of elements provided in the FPGA. The overall slice usage can be seen in the *Logic Distribution* section of Table 6.7. The number of occupied slices is 307 out of 5,888 which is 5% of total slices provided in the FPGA.

We conclude that the major change in resource consumption in an FPGA for debugging hardware depends on memory modules (including the debugging hardware's memory unit and FIFOs). The resource consumption in these memory modules could reduce overall performance of the GPEAT system since they vary according to chromosome size and gene size. As of now, GPEAT is capable of running a variety of EC applications in hardware satisfactorily [26] and the percentage of resource consumption of debugging hardware is relatively low for modern FPGAs.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

Evolutionary computation has been widely used to solve many problems. Much research was conducted in implementing EC systems on hardware using reconfigurable platforms such as FPGAs and Microcontrollers [21-24]. Flexible platform for EC implementation is also described in [24]. Our research concentrates on implementing a flexible reconfigurable platform for EC called GPEAT (General Purpose Evolutionary Algorithm Testbed). The main objective is to allow scientists or engineers with minimal hardware knowledge to implement an application on EC quicker and easier than if they had to build the system from scratch. The architecture and implementation of the GPEAT core is described in [19]. In this thesis, we describe the architecture of the debugging hardware which works with the GPEAT core to further facilitate design and implementation of EC systems in hardware. The GUIs used to configure GPEAT core (Parameter entry GUI) and debug the core (Debugging GUI) are also described.

GPEAT enables quick implementation of a variety of applications that use evolutionary computation on hardware. Common operations in EC can be done faster on hardware with lower clock speed. The parameter-entry-GUI provides a friendly interface through which parameters for different algorithms can be specified. The GPEAT core is configured according to the parameters provided from the parameter-entry-GUI. GPEAT also provides a debugging interface for users to so that they may observe the evolutionary process going on inside the hardware. The debugging hardware communicates with the GPEAT core in order to gather information on the EC being run. The communication allow the debugging hardware to gather data on chromosomes which is then sent back to the computer for performance analysis and debugging purposes. The debugging GUI provides a friendly environment for debugging. Chromosomes and their fitness values are graphically displayed which helps the user in analyzing the design's performance.

7.2 Future work

To be able to achieve our ultimate goal, we plan to make some improvements for the next version of GPEAT. The next version of GPEAT would be more flexible, dynamic and support a broader range of applications. The GPEAT storage system (memory block) can be improved greatly by polishing the memory block in GPEAT core by reducing chromosome storage regions. The current version of GPEAT stores chromosomes categorized by the range of their fitness values in 3 memory regions (each region is called 'bin'). Implementing three bins for storing chromosomes reduces time for sorting process but it also consumes more hardware resources. This is because not all chromosomes are stored in a single bin. Storing chromosomes in single partitioned bin frees up hardware resources but the speed of sorting may be increased. The tradeoffs need to be evaluated. Further improvement to the GPEAT system can be achieved by introducing pipelining. In this version of GPEAT, after a chromosome has been processed from reproduction block and is moving to fitness evaluation block, the reproduction block stays idle until the chromosome is fully processed. Introducing pipelining in this situation would help speed up the system. With pipelining, the reproduction block would process the next chromosome while the first chromosome is being processed in the fitness evaluation block.

A sensor block will be added to GPEAT system providing channels for interfacing with external circuits. Additional buffering and converting circuitry will be required for the sensor bank. The sensor block will provide options for interfacing with digital sensors or analog sensors. Analog-to-digital converter (ADC) will be required to interface with an analog sensors. The parameter-entry-GUI will need to accommodates these new options by providing a friendly interface for users to set up all sensors. Adding the sensor block will make GPEAT more interactive with the operating environment. The objective function used to evaluate chromosomes will be dynamically functioning rather than a fixed function.

In the first version of GPEAT, the user can only specify a rule-based fitness evaluation where multiple rules are used to categorize chromosomes into ranges of fitness values. The ability to specify an objective function in mathematical form (equation) is one of the major keys to make GPEAT more flexible and generalized. In the next version, we are planning to make the parameter-entry-GUI able to parse mathematical expressions that the users enter and convert the expression into VHDL code. Supporting more specific kind of rule-based objective functions is one of objectives for next version of GPEAT. We aim to release GPEAT as an open-system to be used in research communities and obtain feedback from groups of users to refine GPEAT specification.

Another objective for the next version of GPEAT is to embed the functionality for synthesizing a design and generating the configuration file for an FPGA in the parameter-entry GUI. This would provide the users a one-for-all application. The benefits of doing so are that FPGAs from many manufacturers could be supported by GPEAT and also that it will reduce the complexity for the user in the processes from generating VHDL to downloading configuration file to programmable devices. Having all the tools through a single interface will enable users with minimal hardware knowledge to implement GPEAT on any FPGA. One more feature that we would like to add to the parameter-entry-GUI is the ability to inspect and verify parameters user have entered for their validity. This features, could be called a 'design rule check', and would be a tool for checking conflicts among parameters or whether each parameter is in a valid range.

When debugging GPEAT, the system speed can be improved by using a universal serial bus (USB) for the debugging channel. USB can theoretically transfer data up to 400 Mbps which is more than 800 times faster than RS-232 connection that we have used in this implementation which has a transfer rate of 460800 bps. So that bigger variety of information choices can be accessed from the computer, more commands will be added to support new types of information packages. For example, GPEAT would keep a record of how each chromosome was reproduced and transfer the record back to DGUI. Information from the sensor blocks might also be needed to understand if an EC system is running correctly. The debugging GUI has to be polished to support the features that would be added to the debugging hardware. Information visualization can also be added to assist users in performance analysis and debugging.

For future versions, the addition of learning abilities would be considered to support complex applications. Supporting more reproduction types would also increase the ability of GPEAT to do a better search. The implementation of the first version of GPEAT showed us that the theory is sound and that there is a chance to move on to the next step for developing a flexible tool for quick implementation of EC in hardware.

REFERENCES

1) M. Gh. Negoita, S. Hintea: Bio-Inspired Technology for the Hardware of Adaptive System: real-world implementations and applications. Springer-Verlag Berlin Heidelberg (2009)

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

3) O'Neill, M., Ryan, C.: Grammatical Evolution – Evolving programs in an arbitrary language Kluwer MIT Press (1992)

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

5) Exploration in Design Space: Unconventional Electronics Design Through Artificial Evolution Adrian Thompson, Paul Layzell, and Ricardo Salem Zebulum

6) Horia-Nicolai Teodorescu, Lakhmi C. Jain, Abraham Kandel: Hardware Implementation of Genetic Algorithm Modules for Intelligent Systems, Physica-Verlag (2001)

7) A Case for Using Minipop as the Evolutionary Engine in a CTRNN-EH Control Device: An Analysis of Area Requirements and Search Efficacy

8) Thomas Komarek and Peter Pirsch, "Array Architecture for Block Matching Algorithm", IEEE Transaction on Circuit 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 Algorithm to Textture 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) 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

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

21) Scott, S.D.; Samal, A.; Seth, S., HGA: A Hardware-based Genetic Algorithm. Proceedings of the Third International ACM Symposium on Field-Programmable Gate Arrays, 1995. FPGA '95, pp. 53-59

22) Karnik, C.; Reformat, M.; Pechycz, W., Autonomous Genetic Machine. Canadian Conference on Electrical and Computer Engineering, 2002. IEEE CCECE 2002, pp. 828-833 vol.2

23) C. Pei-Yin, C. Ren-Der, C. Yu-Pin, and S. Leang-San, "Hardware Implementation for a Genetic Algorithm", IEEE Transaction on Instrumentation and Measurement, vol. 57, no. 4, April 2008.

24) Pimery, J.; Kumhom, P., Development of A Flexible Hardware Core for Genetic Algorithm. Conference on Intelligent Computing and Intelligent Systems, 2009. ICIS 2009. IEEE International, pp. 867-870

25) R.C. Cofer, Ben Harding: Rapid System Prototyping with FPGAs, Newnes (2006)

26) Tati**,** Kiran Kumar, General Purpose Evolutionary Algorithm Testbed. Master thesis submitted to the faculty of the Universoty of Missouri—Columbia, 2009.

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

28) Kuen-Jong Lee; Si-Yuan Liang; Su, A., "Low-Cost SOC Debug Platform Based on On-Chip Test Architectures" SOC Conference, 2009. SOCC 2009. IEEE International, 2009, pp.161-164

29) Schultz, K.; Paranjape, K., SOC Debug Challenges and Tools. Conference on Very Large Scale Integration, 2006 IFIP International, pp.385-390

30) IEEE Standard Testability Method for Embedded Core-based Integrated Circuits, IEEE Std 1500-2005

APPENDIX

See code available at the University of Missouri website

https://mospace.umsystem.edu/xmlui/handle/10355/3986

VHDL Codes (Developed with Xilinx ISE Version 10 and ModelSim XE-III) for:

Debuggin Hardware (List of files for VHDL codes)

1. debouncer.vhd
2. dlatch.vhd
3. example_RYB.vhd
4. FIFO.vhd
5. GPDC.vhd
6. GPEAT1.vhd
7. GPINC.vhd
8. icontroller.vhd
9. interface.vhd
10. memory.vhd
11. reference.vhd
12. rx_interface.vhd
13. rxicontroller.vhd
14. tx_latch.vhd

Java Codes (Developed with Eclipse Platform Version: 3.4.1 Build id: M20080911-1700) for:

Parameter-entry GUI (List of files for Java codes)

1. DefaultOutput.java
2. DRC.java
3. FitnessIFRule.java
4. GPEATData.java
5. InitialPopulation.java
6. Project.java
7. DebugFrame.java
8. DebuggerFrame.java
9. DefaultOutputFrame.java
10. DefineSystemFrame.java
11. WarningDialog.java
12. DRCFrame.java
13. FitnessIFFrame.java
14. RuleNameFrame.java
15. InitialPopulationFrame.java
16. MainFrame.java
17. ProjectFileFrame.java
18. NumericTextField.java
19. ProjectFile.java

Debugging GUI (List of files for Java codes)

1. Data_Manipulation.java
2. GPEAT_Chromosome.java
3. GPEAT_Info.java
4. GPEAT_Population.java
5. Memory.java
6. MainFrame.java
7. CommInterface.java
8. CommSetting.java
9. SpreadSheetDialog.java
10. BrowserPanel.java
11. NumericTextField.java

GPEAT Encoder (List of files for Java codes)

1. Enc.java
2. FileComponent.java
3. FileGrouping.java
4. Symbol.java
5. SymbolFrame.java
6. TXTEncoder.java