IMPLEMENTATION OF A GENETIC ALGORITHM ON A VIRTEX II PRO FPGA

Michael Vavouras

Technical University of Crete

Department of Electronics and Computer Engineering

Committee

Assistant Professor Ioannis Papaefstathiou (*Supervisor*)

Associate Professor Dionisios Pnevmatikatos

Professor Apostolos Dollas

Chania 2008

Περίληψη

Οι γενετικοί αλγόριθμοι (ΓΑ) είναι αλγόριθμοι εύρεσης-βελτιστοποίησης που βασίζονται στην θεωρία εξέλιξης του Δαρβίνου. Χρησιμοποιούνται ευρέως στην σημερινή εποχή αφού υπερτερούν σε σχέση με άλλους ευριστικούς αλγορίθμους γιατί α) ψάχνουν σε έναν πληθυσμό από λύσεις, β)δεν χρειάζονται να γνωρίζουν λεπτομέρειες για τις παραμέτρους του προβλήματος που καλούνται να επιλύσουν, και γ)κατά την εκτέλεσή τους χρησιμοποιούν κανόνες μετάβασης που βασίζονται σε πιθανότητες και όχι σε ντετερμινιστικά μοντέλα. Επίσης επιλύουν πολύπλοκα προβλήματα γρηγορότερα από άλλους αλγορίθμους γιατί υποστηρίζουν παραλληλισμό.

Στην παρούσα διπλωματική εργασία σε πρώτη φάση πραγματοποιήθηκε εκτενής έρευνα των ΓΑ που έχουν υλοποιηθεί σε FPGAs. Στη συνέχεια αναλύθηκαν χαρακτηριστικά όπως ο χρόνος εκτέλεσης των αλγορίθμων, οι πόροι που καταλαμβάνουν στο υλικό που υλοποιήθηκαν, και έγινε σύγκριση σε σχέση με τις υλοποιήσεις σε λογισμικό.

Στην συνέχεια χρησιμοποιήθηκε ένας ΓΑ που η σχεδίαση του σε υλικό υπήρχε αλλά είχε υλοποιηθεί για παλαιότερη πλατφόρμα και μόνο για τη βελτιστοποίηση της συνάρτησης F(x)=2x. Αφού έγιναν οι κατάλληλες αλλαγές ο αλγόριθμός υλοποιήθηκε για την εμπορική πλατφόρμα XUP που περιέχει την Virtex II Pro FPGA. Στην πλατφόρμα αυτή χρησιμοποιήθηκε ο ενσωματωμένος επεξεργαστής IBM PowerPc για την επικοινωνία με τον υπολογιστή μέσω σειριακής θύρας RS-232 καθώς και για την επικοινωνία με το υλικό στο οποίο υλοποιήθηκε ο ΓΑ. Αφού πιστοποιήθηκε η ορθότητα της υλοποίησης, στη συνέχεια προστέθηκαν περισσότερες συναρτήσεις της μορφής $F(x)=ax^3+bx^2+cx+d$. Για τη σχεδίαση χρησιμοποιήθηκε το εργαλείο Xilinx ISE 7.1 και για την προσομοίωση το Modelsim SE 6.2. Τέλος ο γενετικός αλγόριθμος κατέβηκε στην πλατφόρμα XUP με το Xilinx EDK 7.1. Το αποτέλεσμα είναι μια πλήρως λειτουργική πλατφόρμα για εκτέλεση πειραμάτων με τον συγκεκριμένο γενετικό αλγόριθμο για βελτιστοποίηση 6 συναρτήσεων.

Αφιέρωση

Στους γονείς μου Στέφανο και Κυριακή, στα αδέρφια μου Δέσποινα και Δημήτρη και σε όλα τα αγαπημένα μου πρόσωπα.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή κ. Ιωάννη Παπαευσταθίου για την επίβλεψη και την καθοδήγησή του κατά τη διάρκεια της εκπόνησης της παρούσας διπλωματικής εργασίας. Επιπλέον, θα ήθελα να τον ευχαριστήσω για τις σημαντικές εμπειρίες που μου προσέφερε κατά τη διάρκεια της εργασίας μου στο Εργαστήριο Μικροεπεξεργαστών και Υλικού.

Επίσης θα ήθελα να ευχαριστήσω τα υπόλοιπα μέλη της εξεταστικής επιτροπής, τον καθηγητή κ Απόστολο Δόλλα και τον καθηγητή κ. Διονύσιο Πνευματικάτο για την συνεισφορά τους σε αυτήν της εργασία.

Ιδιαίτερα θα ήθελα να ευχαριστήσω τον Κυπριανό Παπαδημητρίου για την επίβλεψη και την πολύτιμη βοήθειά του στην διπλωματική μου εργασία. Θα ήθελα, επίσης, να ευχαριστήσω τον Δημήτρη Μεϊντάνη που ήταν πάντα πρόθυμος να προσφέρει τη βοήθειά του όποτε χρειαζόταν. Επίσης ευχαριστώ όλους τους προπτυχιακούς και μεταπτυχιακούς φοιτητές του εργαστηρίου.

Ευχαριστώ, επίσης, όλους τους φίλους μου, τον Αλέξη, τον Χρήστο, τον Γιάννη, τον Γιώργο, κλπ. (ξέρετε ποιοι είστε!), για τη συμπαράσταση και την ηθική υποστήριξή τους όλα αυτά τα χρόνια των σπουδών μου στα Χανιά.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου, που με στήριξε και με στηρίζει, όχι μόνο στις σπουδές μου, αλλά σε όλους τους τομείς της ζωής μου.

Contents

Chapter 1

Introduction

1.1. An Introduction on Genetic algorithms

Genetic algorithms (GAs) are search-optimization techniques based on Darwin's theory about evolution. They were invented by John Holland at University of Michigan in the early 1970s [10]. Genetic algorithms are simple to implement and they can solve complex problems in contrast to other heuristic algorithms because they:

- Search from a population of points and not a single point. Most other algorithms can only explore the solution space to a problem in one direction at a time, and if the solution they discover turns out to be suboptimal, there is nothing to do but abandon all work previously completed and start over. However, since GAs have a population of points, they can explore the solution space in multiple directions at once. If one path turns out to be a dead end, they can easily eliminate it and continue work on more promising avenues, giving them a greater chance each run of finding the optimal solution.

- Use objective function - otherwise denoted as fitness function - information and not other auxiliary knowledge. GAs knows nothing about the problems they are deployed to solve. Instead of using previously known domain-specific information to guide each step and making changes they make *random* changes to their candidate solutions and then use the fitness function to determine whether those changes produce an improvement.

- Use probabilistic transition rules and not deterministic rules. The search-optimization process is based on probabilities which give flexibility to the genetic algorithm. If the problem has changed, the genetic algorithm can solve it because it doesn't based on standard parameters but based on probabilities.

- Work with the coding of the parameter set and not the parameters themselves. Many real-world problems cannot be stated in terms of a single value to be minimized or maximized, but must be expressed in terms of multiple objectives, usually with tradeoffs involved. GAs manipulates many parameters simultaneously.

These four characteristics make them powerful, flexible and robust.

Genetic algorithm is a stochastic technique with simple operations based on natural selection. The basic operations are selection, crossover, and mutation. Initially, we have a randomly generated population of candidate solutions of the problem in order to cover the entire range of possible solutions. A **fitness function** selects the parents by evaluating each member's fitness value. The selection of the individuals is performed according to their fitness values. The fittest member has more chances to be selected. Genetic operations such as **crossover** and **mutation** are applied on the parents and the new individuals are generated, called children. Finally a substitution between the old and the new population is made. The algorithm runs until a termination condition is met, number of generation. An indicative flowchart of a genetic algorithm is presented in Figure 1 [25].

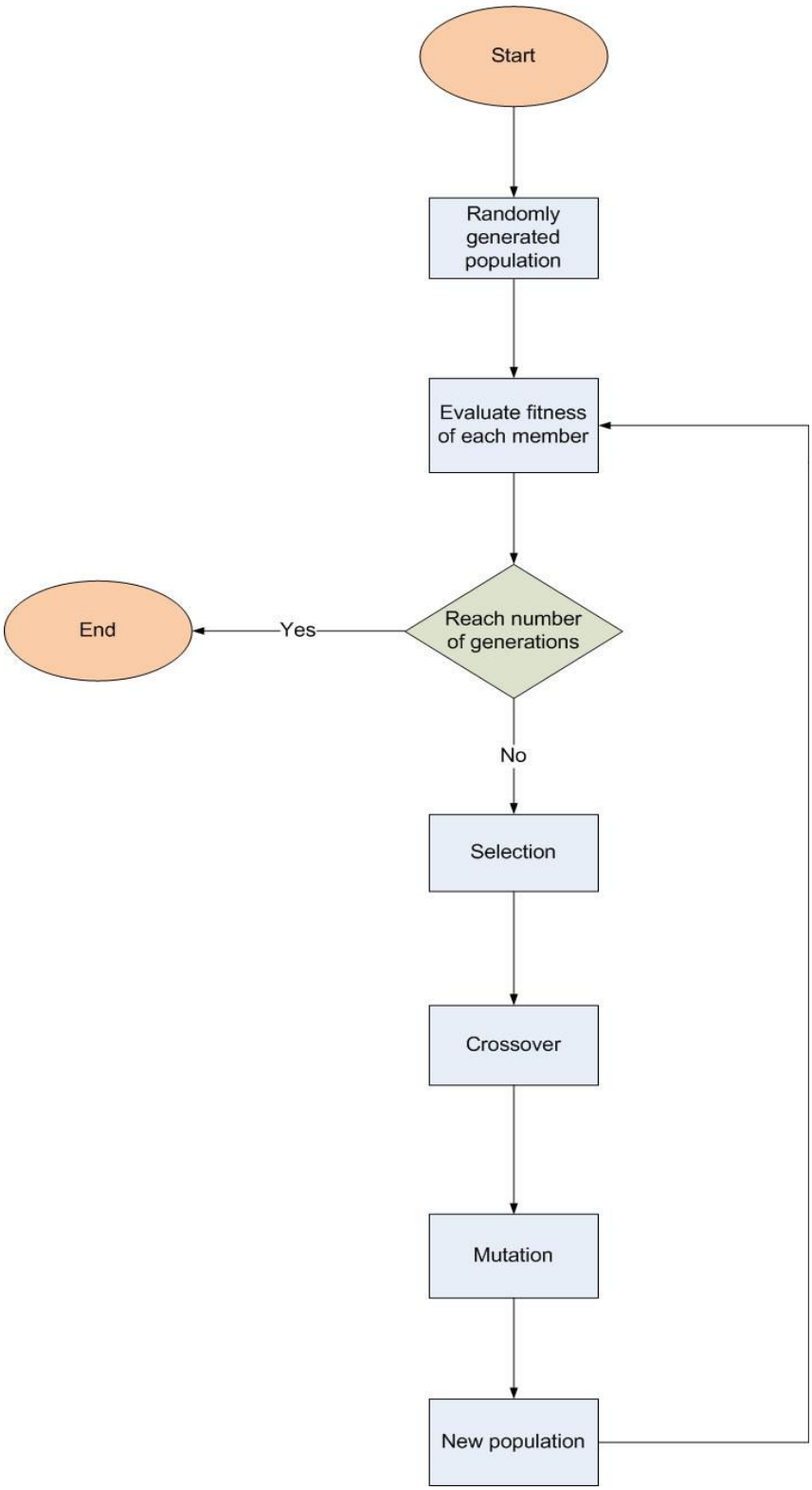

Figure 1: Generic – Type flowchart of a Genetic Algorithm

The basic steps of a genetic algorithm are:

1) **Start:** Generate random population of *n* chromosomes (suitable solutions for the problem).

2) **Fitness:** Evaluate the fitness of each chromosome *x* of the population according to a fitness function *F(x)*.

3) **Selection:** Select two parent chromosomes from a population according to their fitness i.e. the better fitness, the bigger chance to be selected.

4) **Crossover:** Crossover the parents to form two new offsprings, or children, according to a crossover probability. If no crossover is performed, offsprings are an exact copy of parents.

5) **Mutation:** Mutate new offspring at each locus (position in chromosome) according to a mutation probability.

6) **New population:** Use the generated population for a new execution of the algorithm.

7) **Check:** If the termination condition is satisfied, **stop** and return the best solution in the current population else go to step 2.

The chromosome should contain information about the solution it represents in some way. The most common way is **encoding** with a binary string. Each bit in this string can represent some characteristic of the solution, or, the entire string can represent a number. The chromosome then could look like this (see Table 1):

Chromosome 1	Chromosome 2
10110100101001	00001110111010

Table 1: Encoding of a chromosome

There are many other ways of encoding which depend on the nature of the problem. Permutation encoding, value encoding and tree encoding are some of the methods used with success in the field of genetic algorithms.

According to Darwin's theory of evolution, the best chromosomes should survive and form the new population. There are many methods of **selection** such as roulette wheel selection, rank selection, steady-state selection, and elitism. In roulette wheel selection method, the chromosomes with better fitness values have more chances to be selected than others with small fitness values. Imagine a **roulette wheel** where all chromosomes from the population are placed on it. Each chromosome has its place on the roulette according to its fitness value (see Figure2). Then a marble is thrown there and selects the chromosome. Chromosome with bigger fitness will be selected more times. The roulette wheel method could not select the best chromosome because the whole process is randomly (spin the roulette is a random operation).

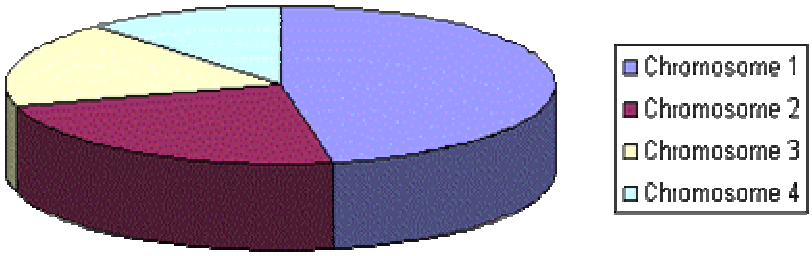

Figure 2: Roulette wheel selection

The previous selection method might face problems when the fitnesses differ very much. For example, if the best chromosome fitness has the 90% of the roulette wheel place, then the other chromosomes that share the 10% of the remaining space will have very few chances to be selected [25].

Rank selection first ranks the population and then every chromosome receives fitness from this ranking. The worst will have fitness ***1***, second worst ***2*** etc. and the best will have fitness ***N*** which equals the number of chromosomes in the population. After

ranking, all the chromosomes have a chance to be selected. But this method can lead to slower convergence, because the best chromosomes do not greatly differ from other ones.

Steady – state selection method is not particular of selecting parents. The main idea of this selection is that a large part of chromosomes should survive and included in the next generation. GA then works in a following way: In every generation a few chromosomes with high fitnesses values are selected for creating a new offspring. Then some bad chromosomes with low fitness value are removed and the new offspring is placed in their place. The rest of population survives in the new generation.

When creating a new population with crossover and mutation, there is a big possibility, that the best chromosome will be lost. **Elitism** is a method, which first copies the best chromosome, or a few best chromosomes to the new population. The rest is done in the classical way. Elitism can very rapidly increase the GA performance, because it prevents from losing the best found solution.

Crossover operation selects genes from the parent chromosomes and creates two offsprings. The simplest way to do this is to randomly choose a crossover point and swap the suffixes of the two parents (see Table 2).

Chromosome 1	11011 | 00100110110
Chromosome 2	11111 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11111 | 00100110110

Table 2: Crossover example

Different types of crossover exist, such as single point crossover, two point crossover, and uniform crossover.

Single point crossover: One crossover point is selected randomly, and binary string from the beginning of chromosome up to the crossover point is copied from one

parent, the rest is copied from the second parent. The second offspring is produced with a similar process. This is shown in Figure 3.

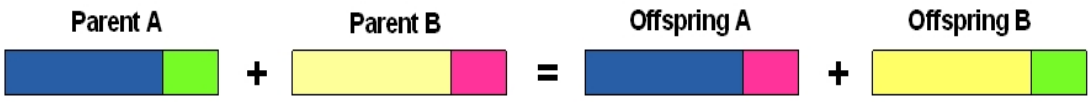

Figure 3: One point crossover

Two point crossover: Two crossover points are selected randomly, a binary string from the beginning of chromosome to the first crossover point is copied from one parent, the part from the first to the second crossover point is copied from the second parent and the rest is copied from the first parent again. This is shown in Figure 4.

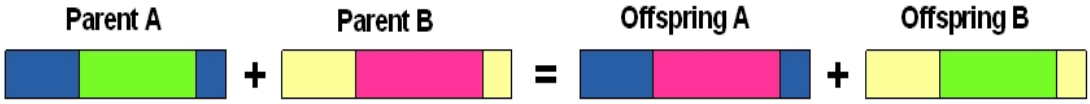

Figure 4: Two point crossover

Uniform Crossover: As shown in Figure 5, bits are randomly copied from the first or from the second parent.

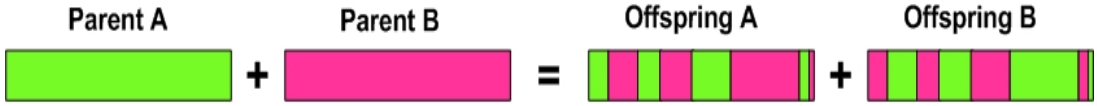

Figure 5: Uniform crossover

After the crossover is performed, **mutation** operation takes place. This is to prevent from falling all solutions in population into a local optimum of solved problem. Mutation changes randomly the new offspring. For binary encoding we can flip a few randomly chosen bits from 1 to 0 or from 0 to 1 (see Table 3).

Original offspring 1	1101111000011110

Original offspring 2	1101100100110110
Mutated offspring 1	1100111000011110
Mutated offspring 2	1101101100110110

Table 3: Mutation example

Genetic algorithms have been applied successfully to many hard optimization problems such as: VLSI layout optimization, job scheduling, function optimization, code breaking, Boolean satisfiability, traveling salesman problem, Hamiltonian circuit problem, bioinformatics, financial.

1.2. Motivation for implementing genetic algorithms on hardware

A Genetic Algorithm is an optimization method based on natural selection. But application of GAs to increasingly complex problems can overwhelm software implementations of GAs, causing unacceptable delays in the optimization process. This is true for any non-trivial application of GAs if the search space is large. Therefore, a hardware implementation of a GA would be applicable for dealing with problems too complex for software-based GAs. Because a general-purpose GA engine requires certain parts of its design to be easily changed, e.g. the function to be optimized, Hardware-based Genetic Algorithm (HGA) can benefit from field-programmable gate arrays (FPGAs). Reprogrammable FPGAs that are programmed via bit patterns stored in static RAMs are powerful candidates for the development of HGA system [10].

A simple empirical analysis of software-based GA's indicated that a small number of simple operations and the function to be optimized were executed frequently during the run. Neglecting I/O, these operations accounted for 80-90% of the total execution time. If m is the population size (number of strings manipulated by the GA in iteration) and g is the number of generations, a typical GA would execute each of its operations mg times. For complex problems, large values of m and g are required, so it is imperative to make the operations as efficient as possible. A work by Spears and De Jong [27]

indicates that for NP-complete problems, m=100 and values of g in the order of 10^4 – 10^5 may be necessary to obtain a good result and avoid premature convergence to a local optimum. Pipelining and parallelization can help providing the desired efficiency, and they are easily implemented in hardware. The nature of GA operators is such that GAs lends themselves well to pipelining and parallelization. For example, selection of population members can be parallelized to the practical limit of area of the chip(s) on which selection modules are implemented. Once these modules have made their selections, they can pass the selected members to the modules, which perform crossover and mutation, which in turn pass the new members to the fitness modules for evaluation. This way a coarse-grained pipeline is implemented. The capability for parallelization and pipelining helps in efficiently mapping GA to hardware [3], [10].

1.3. The selected hardware genetic algorithm

After making a research for genetic algorithms implemented on FPGAs, we decided to occupy with the hardware based genetic algorithm proposed by Scott [10]. The main reason was that the VHDL implementation was available through the Internet [26]. The initial design was implemented under the Mentor framework targeting a BORG board with Xilinx FPGAs. We ported the design to a modern platform, XUP, and verified its correct operation.

1.4. Contributions of this work

The contributions of this thesis are the following:

- Implementation of a genetic algorithm on a Virtex II Pro FPGA platform.
- VHDL coding and synthesis with Xilinx ISE 7.1 tools, post place and route simulation and verification with Modelsim 6.0. In addition, Xilinx EDK 7.1 was used for including the embedded PowerPC processor.

- Six fitness functions were implemented for extension as compared to one fitness function implemented on the initial implementation.

- Results evaluation from a different point of view compared to the initial implementation.

Chapter 2

A survey of genetic algorithms and their implementations

2.1. Introduction

Genetic algorithms are known from the beginning of 90's for their assistance in new technologies. They are simple to implement and they can solve complex problems in contrast to other heuristic algorithms. They use techniques such as selection, crossover and mutation as they were described in Chapter 1.

2.1.1. A genetic algorithm example

As a simple example, imagine a population of four strings having five bits each. Also imagine a fitness function F(x) = 3x which simply returns the integer value of three times the binary integer (e.g. F (00000) = 0, F (00001) = 3, F (00010) = 6, etc.). The goal is to optimize in this case maximize the fitness function over the domain 0 < x <

31. Now imagine a population of the four strings generated randomly before genetic algorithm starts. The corresponding fitness values and percentages come from the fitness function F (x) [10], [23].

In the following Tables, 4 and 5 the "% of Total" column contains the probability of each string's selection. So initially 11010 has 41.2 % chance of selection, 01101 has 20.6 % chance, and so on. The selection process can be thought as spinning a "weighted roulette wheel" like in Figure 2. The results from the spins are given in the "Actual Count" column of Table 4. As expected these values follow the corresponding values of the "Expected Count" column.

After selecting the strings the genetic algorithm randomly pairs the newly selected members and looks at each pair individually. For each pair, e.g. A = 11010 and B = 01101, the genetic algorithm decides whether or not to perform crossover. If it does not, then both strings in the pair are placed into the population with possible mutations. If it does then a random crossover point is selected and crossover proceeds. Then the children A' and B', are placed in the population with possible mutations. The genetic algorithm invokes the mutation operator on the new bit strings very rarely usually in the order of less than 0.01 probability, generating a random number for each bit and flipping that bit if the random number is less than or equal to the mutation probability.

After the current generation's selections, crossovers and mutations are completed, the new strings are placed in a new population representing the next generation. In this example generation, average fitness increased by 38 % and maximum fitness increased by 11%. This simple process would continue for several generations until a termination criterion is met.

i	String (Binary) xi	String (Decimal) xi	Fitness F(xi)= 3xi	% of Total F(xi) / Σfi	Expected Count F(xi)/(mean)F	Actual Count
1	11010	26	78	0.412	1.65	2
2	01101	13	39	0.206	0.82	1
3	10110	22	66	0.35	1.39	1
4	00010	2	6	0.031	0.12	0
	Sum (ΣF)		189	1	3.98	4
	Avg (mean F)		47.25	0.25	0.995	1
	Max		78	0.412	1.65	2

Table 4: Four strings and their fitness values

i	After selection	Mate	Crossover Point	After Crossover	Fitness F(x) = 3x
1	11|010	x_3	2	11101	87
2	1|1010	x_4	1	10110	66
3	01|101	x_1	2	01010	30
4	1|0110	x_2	1	11010	78
	Sum				261
	Avg				65.25
	Max				87

Table 5: The population after selection and crossover

2.2. Implementations of genetic algorithms on FPGAs

In this section we present 18 implementations of genetic algorithms on FPGAs. We present the current status of genetic algorithms in FPGAs and give an overview of the existing approaches and their trade offs.

2.2.1. Koonar et al

Koonar et al. proposed a genetic algorithm for circuit partitioning in VLSI physical design automation. The design consists of 9 modules including the external memories (see Figure 6) [3].

The control registers are loaded with appropriated values using the CPU interface and the process starts with an active high pulse. After the process starts, the system takes the netlist from the Top-level IO's and stores it into the netlist memory. Then the core of the system generates the initial population randomly and stores it into the chromosome memory. The selection module chooses parents with good fitness values and sends the addresses in the crossover and the mutation module. In this module the two functions of crossover and mutation performed on the parents and the new population is generated. The fitness module is responsible for creating the fitness values for the children. Main controller checks the modules by sending control signals. The selection module uses Tournament selection, choosing the one with the best fitness value for crossover and the children are in the chromosome memory. The selection module reads four random fitness values from the fitness memory. It compares two pairs of fitness values and chooses the best from each pair. The addresses of these two best values stored until the selection module will be enabled again. These addresses represent the starting address of the parents in the chromosome memory. The chromosome memory is separated in two parts. Parents stored in the first part and children in the other. One word from each parent is read from the memory and a counter is increased for the words of the chromosomes. The crossover module creates a random crossover mask for each word of the parents. The crossover and mutation rates compared with a random 8-bit number. If this number is smaller than the crossover and mutation rates these operations are performed, otherwise the parents are copied to the children. The results are stored word by word in the memory. The starting memory address of the children obtained from the main control state machine and the process repeated until the counter reaches the length of the chromosome. When the population is generated, the fitness module creates values for the children. The fitness module defines for each net if the partitioning of the chromosome

generates a cut. For each chromosome 0, the fitness counter becomes 0 and the chromosome and the net are read word by word from the chromosome and the net memory. Moreover, for each word of the chromosome and the net, an 'AND' operation followed by an 'OR' operation. This generates information, based on the present word of the chromosome, in which partition the net belongs. If a net is met in a specific partition the bit it indicates this is kept. If both of the bits are '1' this shows a cut and the fitness counter increments. This is repeated for each word and the final fitness is modified in comparison with the number of cells that exists in each partition.

The main controller starts reading the input netlist, after receives a signal and loads it into the netlist memory. Then it generates random chromosomes and random population in the chromosomes memory. After initializing the memories, three functions are executed by the main controller. Fitness, selection and crossover and mutation function, using some control signals to produce the final population.

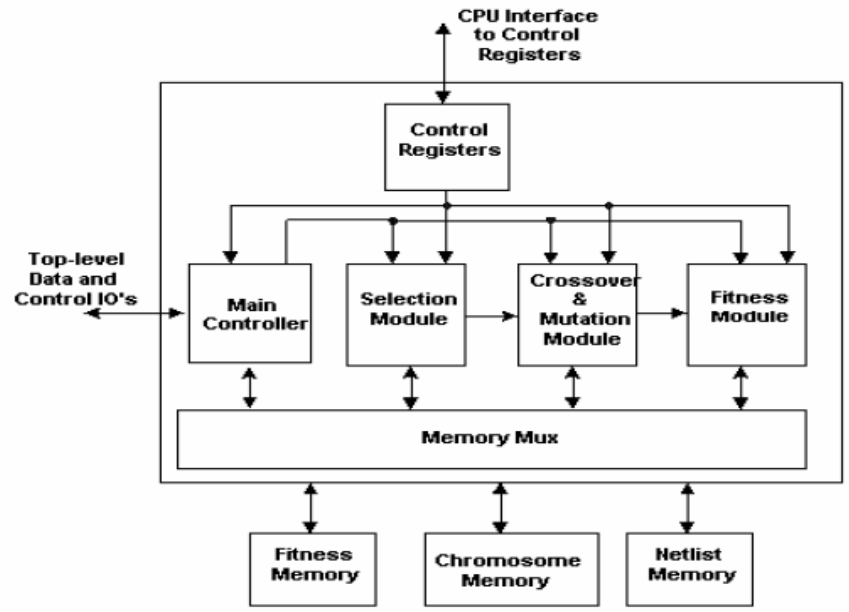

Figure 6: Architecture of the genetic algorithm processor

We present Tables 6, 7, 8 and 9 with results below for hardware and software implementations for different generation count and population size:

Name	Number of nets	Number of modules
Pcb1	32	24
Chip1	294	300
Chip3	239	274

Table 6: Benchmarks

Benchmarks	Generation Count	Software Time(ms)	Hardware Time(ms)
Pcb1	20	200	1.63
Nnets=32	60	600	4.91
Nmods=24	100	900	7.20
Chip1	20	1,700	40.5
Nnets=294	60	4,800	121.25
Nmods=300	100	8,100	202.32
Chip3	20	1,200	23.23
Nnets=239	60	3,400	69.52
Nmods=274	100	5,900	116.23

Table 7: Performance results for Hardware GA and Software GA for different Generation Count

Benchmarks	Generation Count	Software Time(ms)	Hardware Time(ms)
Pcb1	20	200	1.63
Nnets=32	60	700	4.82
Nmods=24	100	1,100	7.20
Chip1	20	1,700	40.5
Nnets=294	60	4,900	122.25
Nmods=300	100	8,800	203.6
Chip3	20	1,200	23.23
Nnets=239	60	3,800	69.36
Nmods=274	100	5,700	115.32

Table 8: Performance results for Hardware GA and Software GA for different Population Size

Parameters	Parameters Value
Population Size	20
Generation Count	20
Crossover Rate	0.99
Mutation Rate	0.01
Crossover Type	Uniform
Mutation Type	Tournament

Table 9: Default genetic parameters

During the simulation results we realize that hardware implementation is much faster than software design.

2.2.2. Tang et al

Wallace et al. implemented a hardware genetic algorithm using FPGA known as FGA. The PCI based hardware GA processor consists of 2 FPGAs. The first is used for the bus interface and the control unit with the implementation of the genetic operators. The second is used for the implementation of an objective function [4]. The block diagram is presented in Figure 7 below:

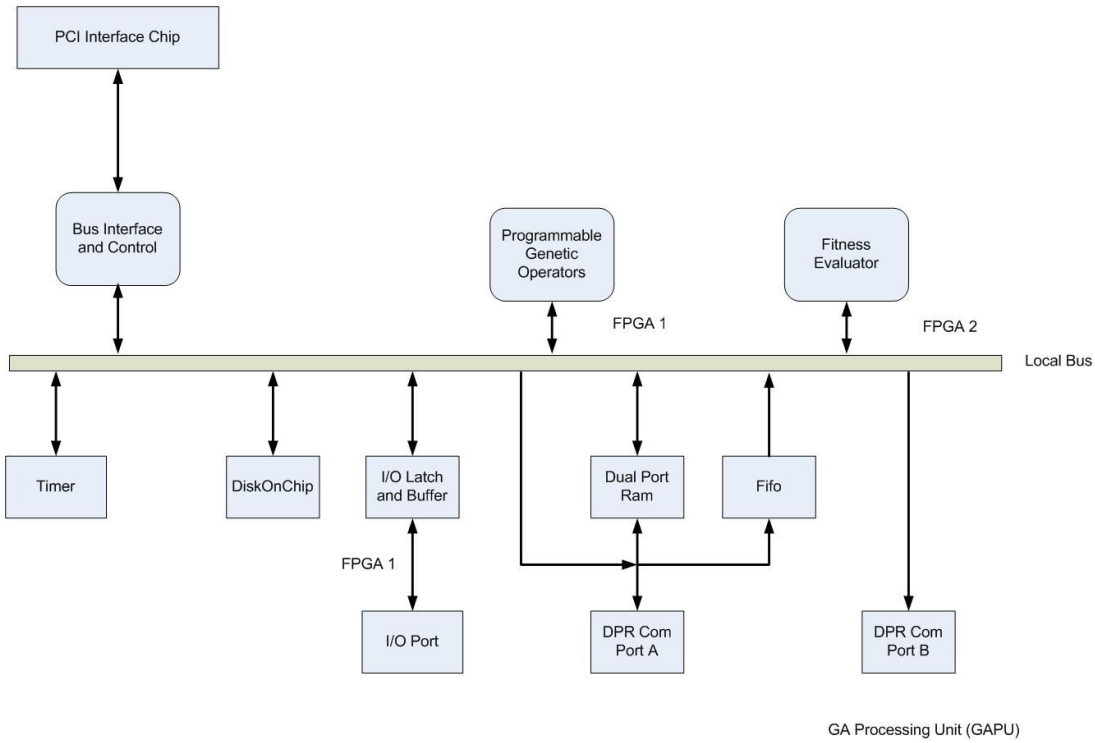

Figure 7: Block Diagram of FGA

First of all, it is a PCI interface with two dual port RAMs. The genetic operations are implemented in hardware and they are programmable. The fitness function can be modified by a single FPGA. In this design, the genetic operators are implemented in parallel and pipelined architectures, using FPGA. The parts of the genetic operators are the pseudo random number generator, the selection module, the crossover module and the mutation stage. The pseudo random number generator consists of a fast asynchronous clock 100MHz and the random number it generates, it is 96 bit length. For the selection module we use roulette wheel selection with optimized hardware. In the crossover module if the random string of bits generated from the random number generator is smaller than the value of the register for crossover and if another register is high than the crossover begins. Two 16-bit registers produced from the parents. There are 3 types of crossover, the one point crossover, the multi point crossover and finally the uniform crossover. There are four types of mutation, the one bit mutation, the multi bit mutation, masked mutation and random mutation. To test this implementation proposed above they used population size of 256. They used the

uniform crossover method with crossover rate 0.9 and the random mutation method with mutation rate 0.09. It took 500 μsec to initialize the population with the single FGA in contrast with 5.34 msec in Pentium 2.4 GHz machine. So we realize that with FGA boards we can achieve linear speedups.

2.2.3. Aporntewan et al

Aporntewan et al. proposed a hardware implementation of the compact genetic algorithm. The compact genetic algorithm is implemented in VHDL and fabricated in FPGA. Moreover, the compact genetic algorithm, represent a population as a vector with l dimension, where *l* is the length of the chromosome. *I* dimension of the vector is the probability to be 1 or 0. So, the compact genetic algorithm manages the vector instead of the population and this decreases the number of the bits demanded to keep the stored population. Consequently, we can use registers for the probability vector. We have the *population (n)*, which represented as an l dimension probability vector and the *chromosome length (l).*The p (i) is the probability of *i* bit and initially is 0.5. Then we have the *a (i)* and *b (i)* which generated according to *p*.

The *a (i)* is 1 with probability *p(i)* and 0 elsewhere. Afterwards, we have the fitnesses for *a, b*. If $f_a \geq f_b$ and if *a (i)* = 1 and *b (i)* = 1, the *p (i)* increased by 1/n, otherwise decreased by 1/n. This continues until the *p (i)* takes the value 0 or 1. Finally *p* gives us the finally solution [2]. The pseudo code for the compact genetic algorithm is shown in Figure 8 below:

```
Compact GA parameters:
n: population size.
l: chromosome length.

for i = 1 to l do
    p[i] = 0.5;

repeat
    for i = 1 to l do
        a[i] = { 1  with probability p[i]
               { 0  otherwise
        b[i] = { 1  with probability p[i]
               { 0  otherwise
    endfor

    // Fitness calculation
    f_a = fitness(a)
    f_b = fitness(b)

    for i = 1 to l do
        if f_a ≥ f_b then
            if a[i] = 1 and b[i] = 0 then
                p[i] = min(1, p[i] + 1/n)
            if a[i] = 0 and b[i] = 1 then
                p[i] = max(0, p[i] - 1/n)
        else
            if a[i] = 1 and b[i] = 0 then
                p[i] = max(0, p[i] - 1/n)
            if a[i] = 0 and b[i] = 1 then
                p[i] = min(1, p[i] + 1/n)
        endif
    endfor
until each p[i] ∈ {0,1}
```

Figure 8: Pseudo code of the compact genetic algorithm

For the random number generator we use one dimensional two state cellular automata. We use 8 bit for the number which is satisfied for a random process. When we have a lot of bits we have best quality. The numbers of the random number generators are the same with the length of the chromosome. The probability register is a module that keeps the *p(i)*, which is an 8 bit integer. The population must be a power of two. The hardware organization shown below consists of comparators which compare two integers and if *m>n* the output is 1, otherwise is 0. In the design below we can see the buffers where they keep the *a(i)* and the *b(i)*. There are also two fitness evaluator modules to compute the values of *a, b*. The hardware organization is shown in Figure 9 below:

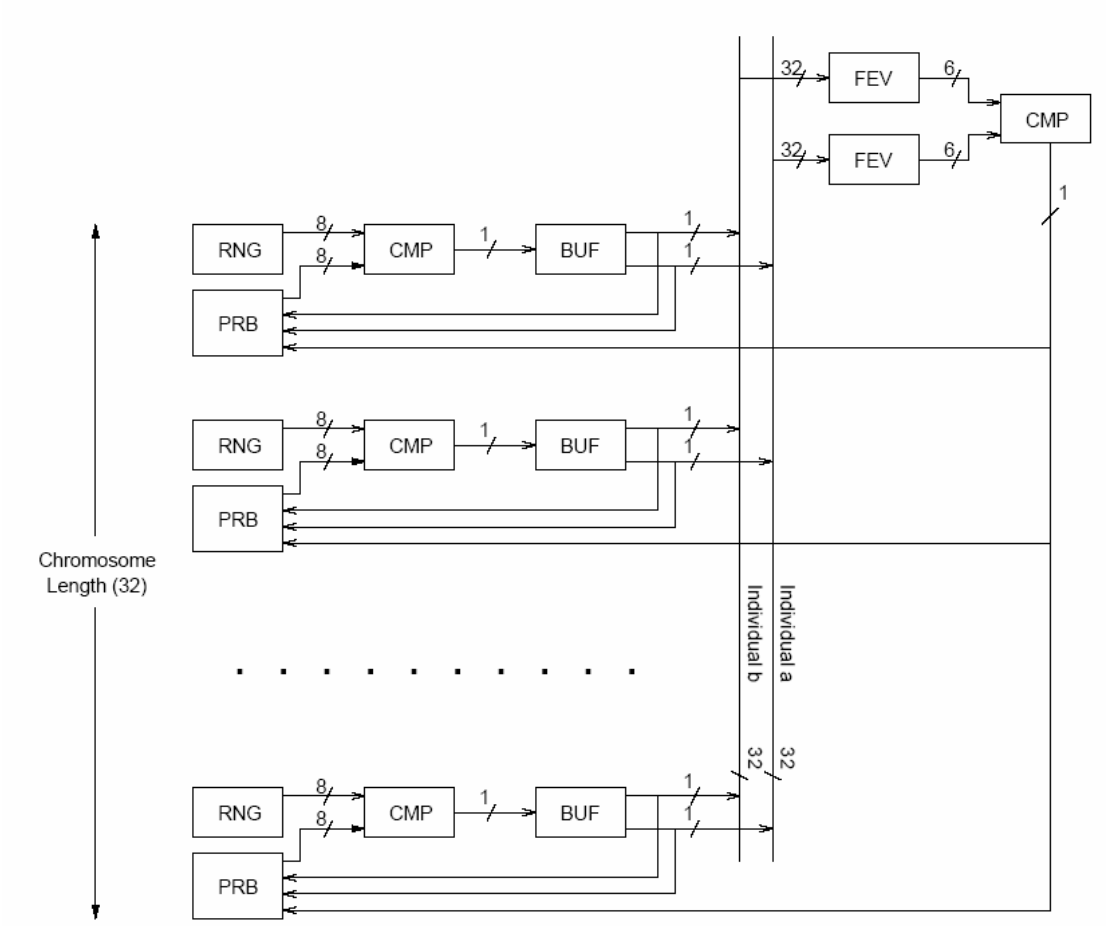

Figure 9: Hardware Organization

The compact genetic algorithm is more suitable for hardware implementations than the simple genetic algorithm. The hardware genetic algorithm is 1000 times faster than a version of software (see Table 10). This compact genetic algorithm is targeted into a Xilinx Virtex V1000FG680 FPGA chip.

Software (200 MHz Ultra Sparc 2)	Hardware (FPGA 20 MHz)	Speedup
2:30 min	0.15sec	1000

Table 10: Comparison between software and hardware

2.2.4. Tommiska et al

Tommiska et al. suggest a genetic algorithm with ALTERA hardware description language implemented in a 10K FPGA. The hardware consists of a Pentium microprocessor with 4 PCI slots. The population is located in the EABs (embedded array blocks), which are flexible RAM. The fitness function is located in the LABs (logic array blocks), which are for arithmetic operations [8]. The random number generator is very important, so they choose a linear shift register (LSHR), which is simple to implement and it generates good random numbers. In this design, there are three LSHR because of the periodicity of the random numbers. The random number generator consists of a noise diode, an amplifier and an analog to digital converter. In the mutation stage, they use one point mutation with rate 1 or different. The genetic algorithm run in a pipeline and consists of four stages, which are separated by register banks. The register banks are used for the synchronization of the pipeline and for the safety of the chromosomes addresses. This means that the memory address we read the chromosome is the same we write it. In the first stage, we select two random chromosomes from the memory. The memory is implemented synchronous with distinctive read and write ports. In the second part of the pipeline the two random chromosomes are submitted for crossover and mutation and they pass through the third stage with the offsprings. The crossover is selected randomly and the offsprings are submitted for mutation, which is implemented as invert of the random selected bit in the 32 bit chromosome. In this implementation Tommiska et al used crossover rate equal to 1 and mutation rate equal to 0.31. The fitness evaluation module has the four chromosomes (parents and offsprings). The fitness function is implemented as a simple comparison of the 32 bit quantities. The chromosomes are compared with round robin algorithm and the number of comparisons is six (one with other three).The best two chromosomes are selected and write back to the last stage of the pipeline. Finally, these two chromosomes are written back to the same memory address, where they had read before. The process described above is presented in Figure 10.

The design proposed here is very fast (4 clock cycles) in comparison with software implementations. FPGAs are faster than microprocessors for a genetic algorithm. The

clock cycle of the entire pipeline is 80 ns. The selection, crossover mutation, fitness evaluation and write back operations spent 160ns. They run the same algorithm coded in C language in a Pentium processor and they realize that the genetic operations took 34 μsec. This means that the implementation they proposed is 212 times faster than software.

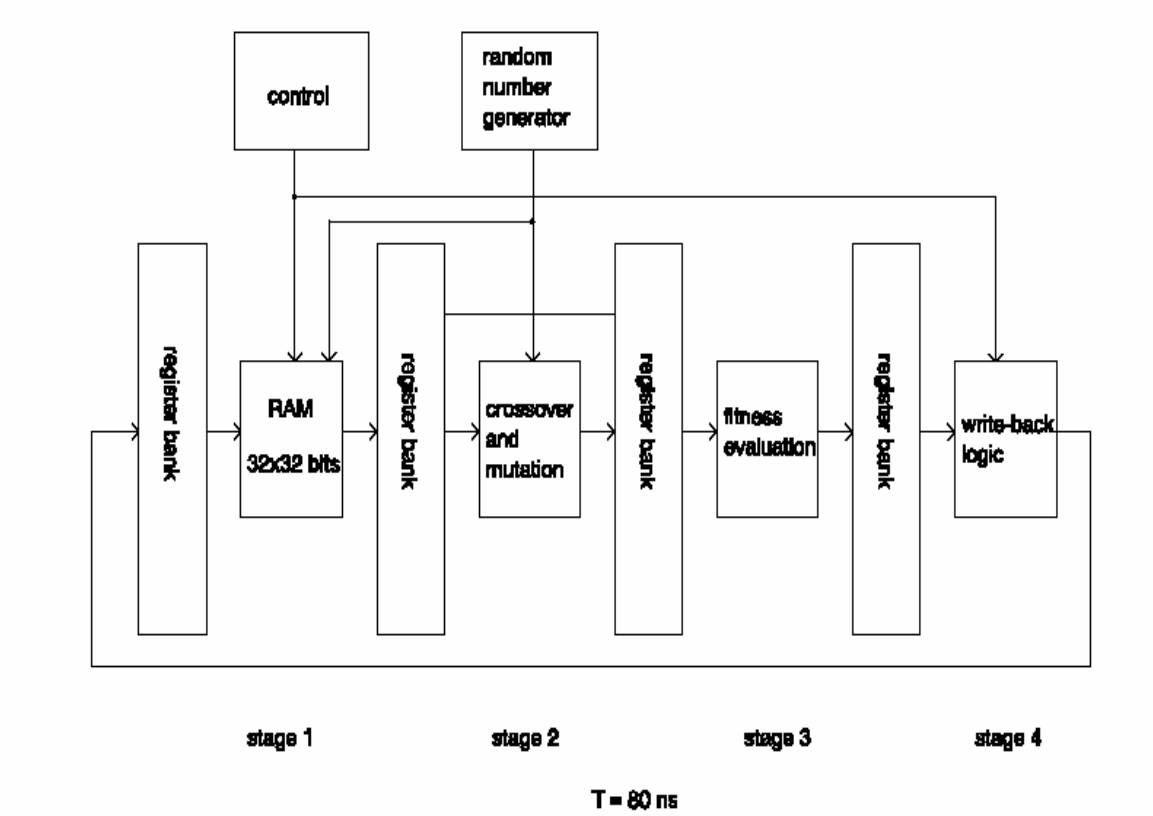

Figure 10: Genetic algorithm pipeline on four stages

2.2.5. Emam et al

Emam et al. introduced an FPGA based genetic algorithm in the application of the blind signal separation. The variables are represented as genes in a chromosome. The natural selection guarantees that chromosomes with best fitness values will be generated in the new population because they result from parents with the best fitness

values. Blind signal separation means that we detect mixtures of independent sources and making use of these signals, we recover the original signals. We can take these mixtures signals instantaneous [11].

$$x(t) = A \times s(t)$$

A: the mixture matrix, s (t): sources vector and x (t): the final vector (without noise). The sources can be separated by finding a matrix w that

$$w \times A = P \times D$$

P: modified matrix, D: diagonal matrix. So the signal we recover is:

$$y(t) = w \times x(t) \Rightarrow y(t) = P \times D \times s(t).$$

The chromosomes of the application composed of the filter factors and the fitness value. Their length is 64 bits, 48 bits for the filter factors and 16 bits for the fitness value. Figure 11 presents the chromosome representation.

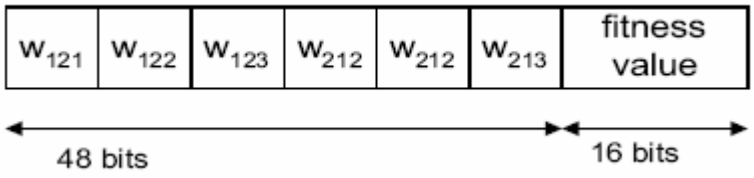

Figure 11: Chromosome representation

Genetic algorithms are responsible for optimizing the factors of the filters, which are used to separate two mixed signals. Initially, the factors of the filters take random values and these values are sent to the filters in the DSP. The output signals are tested for separation and evaluated by the fitness function. The fitness function represents the similarity between the real and the estimated model.

$$F = 1 - e(k) \quad \text{and } e(k) = y(k) - y'(k)$$

The factors of the filters and the fitness values are placed in a matrix with the probability of mutation, in order that registers will be generated in the FPGA. The genetic operator will choose the chromosomes through the selection. The operations of crossover and mutation will recombine the chromosomes to take the offspring. The new generation goes back to the DSP processor so that the output signals will be estimated for the separation. This system described above implemented in a PCI board consists of a DSP processor and an FPGA. The size of the population is 80 and the mutation probability is 0.1. Figure 12 presents the process we described. The results are shown in Table 11. Hardware genetic algorithm is a very good implementation for real time application where the time of the whole application depends on the fitness function.

Block Name	Total Number of Slices	Number of Equivalent Design Gates	Number of IOB Gates	Maximum Path Delay (ns)	Maximum Frequency (MHz)
RNG	41	721	1,776	15.67	173.13
Selector	440	5,264	1,392	63.366	62.449
Crossover	193	2,354	1,392	27.138	63.291
Mutation	148	1,834	1,392	38.79	83.486
Total GO	1,081	12,618	1,392	69.789	58.782

Table 11: Results

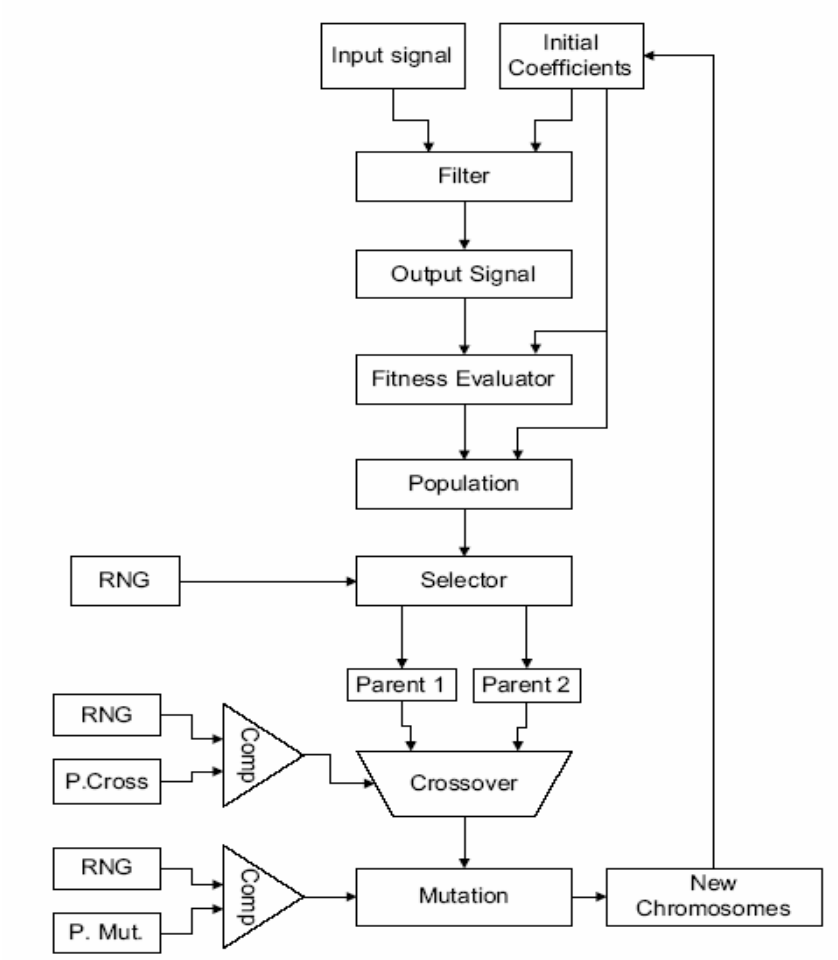

Figure 12: Genetic algorithm in Blind signal separation

2.2.6. Scott et al.

Scott et al. suggested a hardware-based genetic algorithm implemented in VHDL [10]. The first part denoted as front-end of the HGA, consists of an interface running on a computer. This interface takes the parameters from the user and writes them in a shared memory standing between the first and the second part, denoted as back-end, of the HGA. User defines the fitness function in C or VHDL. All of the modules have been written in behavioral non synthesizable VHDL except the memory [10], [23].

Initially, all the parameters are loaded into the shared memory. The memory module plays the role of the HGA controller. It receives a signal of the front end part

and connects the interface with the back-end. The memory interface module updates fitness module, crossover and mutate module, random number generator and the population sequencer that the HGA starts. Each module mentioned above asks for the parameters it needs from the memory module. The population sequencer initiates the pipeline operation, asking from the memory module the members of the population to pass them through the selection module. The selection module takes the members from the population sequencer and decides for them until a suitable pair come. When the pair passes to the crossover and mutation module, the selection module resets itself and restarts the selection process. When the crossover and mutation stage takes the pair from the selection module, decides if it makes crossover and mutation and when it finished the offsprings are sent to the fitness module. In the fitness module the two members are evaluated and are written to the memory interface. Then, the fitness module continues its work and when it finished sends a signal to the front end.

The design is coarse – grained pipeline. When a module finishes a process, waits for another input to continues. So, genetic functions are not good to be interrupted if others are running. The parallelism can be inserted putting two selections modules. The population sequencer is the most time – consuming part of this design. Scott et al simulated the system for several fitness functions and they have proven its speedup over the software approach. An extended presentation of this work is available in Chapter 3.

2.2.7. Mostafa et al

Mostafa et al. proposed an implementation of a parallel – pipelined hardware genetic algorithm using VHDL for programming the FPGA. The proposed pipelined – parallelized hardware genetic algorithm (PPHGA) consists of five parts. First of all, the random number generator supplies the system with all the random numbers it needs. They suggested linear cellular automation technique for the pseudo random bit – strings. There are 16 different bit – strings and the clock cycle is huge. In this way, they

have more randomness than the linear feedback shift register. The output of the RNG supplies the selection module and the crossover and mutation module. The sequencer module is used for drawing the members of the population in sequence from the memory due to a protocol, which is used for all the methods. After the sequencer module there is the selection module, which selects the members that are going to be crossover and mutated. The fourth module is crossover and mutation, which is responsible for the crossover and the mutation of the bit – strings. The fifth module is the fitness module which calculates the fitness values of the members and decides if the program must be stopped [9].

There is a 32bit floating point module to make operations with the decimal point. Finally, there is the memory and the control unit which organizes the process between the modules and the memory. Moreover, Mostafa et al used three 32 bit floating point registers: BR, AC, QR. Each register is separated in two parts. The registers give delay in the design and the biggest delay is observed in the fitness module. The PPHGA has practical applications which are the linear function interpolation, the thermistor data processing and the computation of vehicle lateral acceleration. In these applications they realize that the PPHGA perform better than other search algorithms.

2.2.8. Peter Martin

Peter Martin presented a hardware implementation of a genetic programming system using FPGAs and Handel – C. Handel – C is a high level programming language which is located in the centre of the hardware. The output of this language is a file that is used for inserting data in the FPGA. The syntax is like C language. The advantage of the parallel hardware generation is that we use hardware and we can achieve parallelization directly. In a computer with simple processor this can be succeeded with the division of time. Handel – C supports parallelization and integers. The communication between hardware and the outside world is accomplished with interfaces. In addition, the expressions take 1 clock cycle and are constructed from combined logic. In other hand,

does not support stack and recursive functions directly. A Handel – C code is portable and it can be used for several times. From the genetic operations only the fitness function is prepared on the FPGA. The other functions such as crossover and mutation are implemented in a host computer. By way of experiment FPGA is 200 times faster than a Pentium processor of 750 MHz. Sometimes, the fitness function and the initial population are implemented in the FPGA. In an experiment they put 4 FPGAs for each operation and utilizing a pipeline each FPGA passes its results to the others. They compared this design with an implementation in software in 125 MHz workstation and they realized that the results were 4 times better. Peter Martin implemented a complete genetic programming system with Handel – C in hardware [1].

Handel – C supports direct parallelization by activating effective implementation of instructions and this increment the efficiency. Random number generator is used twice. Firstly, to generate the initial population and secondly to choose which of the genetic operators is going to be used. The linear feedback shift register was used and the word length was 32. The pseudo random number is generated in one clock cycle. In the breeding policy, the tournament selection was used and the mutation probability was 0.1, the crossover probability equal to 0.7 and the copy of the individuals was 0.2. The mutation function can change zero, one or more contents of the instructions. Thereafter, the crossover operator copies segments from one program to another. Peter Martin runs two problems in 2 different environments. The first problem is the regression which Peter Martin runs it in the power PC and in the Handel-c environments (see Table 12).

Measurement	Power Pc Simulation	Handel-C (Single fitness evaluation)	Handel-C (4 parallel fitness evaluation)
Cycles	16,612,624	351,178	188,857
Clock Frequency (MHz)	200	25	19
Estimated Gates	n/a	142,443	228,624
Number of Slices	n/a	4,250	6,800

Percentage of slices used	n/a	22%	35%
Speedup (cycles)	1	47	88
Speedup(time)	1	6	8

Table 12: Results from the regression problem

Furthermore, Peter Martin runs the XOR problem in the previous two environments and the results he realized are shown below in the Table 13.

Measurement	Power Pc Simulation	Handel-C (Single fitness evaluation)	Handel-C (4 parallel fitness evaluation)
Cycles	27,785,750	715,506	384,862
Clock Frequency (MHz)	200	22	18
Estimated Gates	n/a	89,205	228,624
Number of Slices	n/a	4,630	7,434
Percentage of slices used	n/a	24%	38%
Speedup (cycles)	1	38	72
Speedup(time)	1	4	6

Table 13: Results from the XOR problem

For the Handel-c simulation and hardware implementation the CELOXICA RC1000 FPGA board was exploited shown in the Figure 13 below.

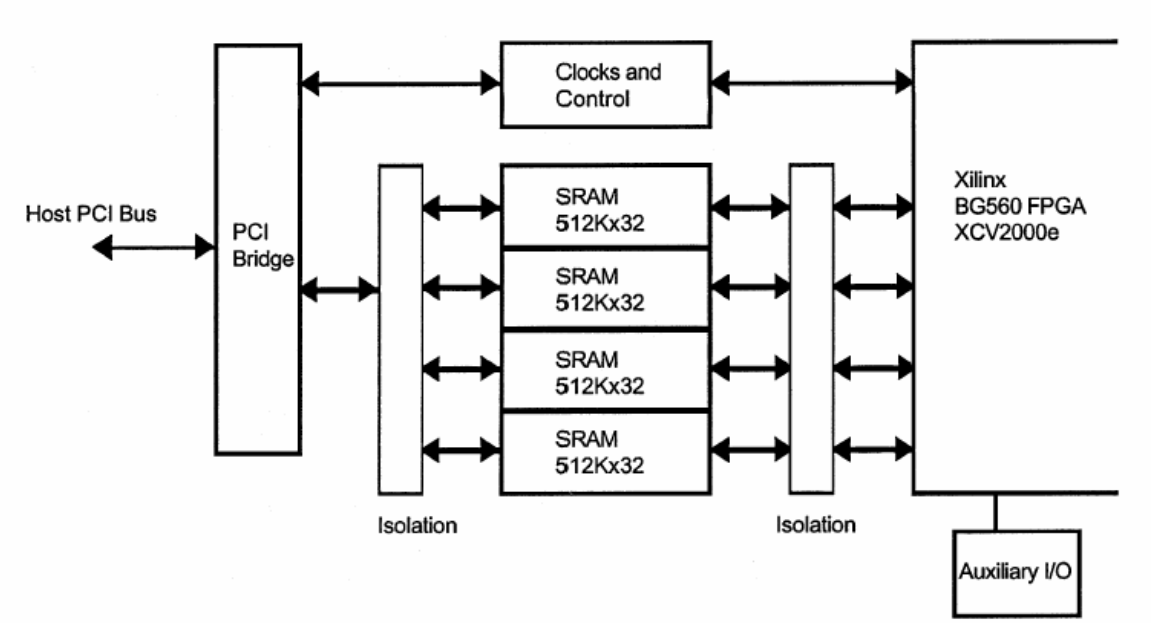

Figure 13: Block diagram of the CELOXICA RC1000 FPGA board

2.2.9. Tachibana et al

Tachibana et al proposed a hardware implementation method of multi objective genetic algorithm. Multi objective genetic algorithms (MOGAs) are techniques to solve multiple objective optimization problems. MOGAs have a set of optimal solutions called pareto and that's why they demand a lot of calculating time. In this method we need to have a variety of individuals because this decrements the comparisons. In the hardware implementation of the single objective genetic algorithm, the crossover and mutation modules must be implemented separately, but their outputs must have the same structure in order to implement the pipeline. At the minimal generation gap model two members are choose from the population and be subject to crossover and mutation. The selection module chooses the member with the best fitness value and replaces it with the worst, in the family of parents. The advantage of this method is that we can benefit from the pipeline and the parallelization and reduces the required memory for the population. The parallel architecture proposed here utilizes the island genetic algorithm. This technique separates the population in several sets. Each of the sets is regarded like an island and is independent. Small part of the population migrates

periodically in order to all the islands cooperating to find the optimal solution. In the proposed method crossover and mutation are the same with the single objective genetic algorithm. The selection module has two operations, the normal and the biased selection. In the first one we compare two parents and choose the best one. After the choice we compare the parent with the offspring and the offspring is replaced with the parent. Biased operation we have when the offspring is compared with all the possible solutions. If there are individuals with the same chromosomes as the offspring, are removed. For the parallel execution of the multi objective genetic algorithm we use the island technique described above with one difference. We keep the diversity of the individuals by letting one island to use the normal selection and the others the biased. The immigration of the individuals chooses from k biased islands one by one in a single period. They are compared and we choose the dominant individuals for immigration in a normal island. The same time one member is choose from the normal island to immigrate to a biased island, after we make it double [6].

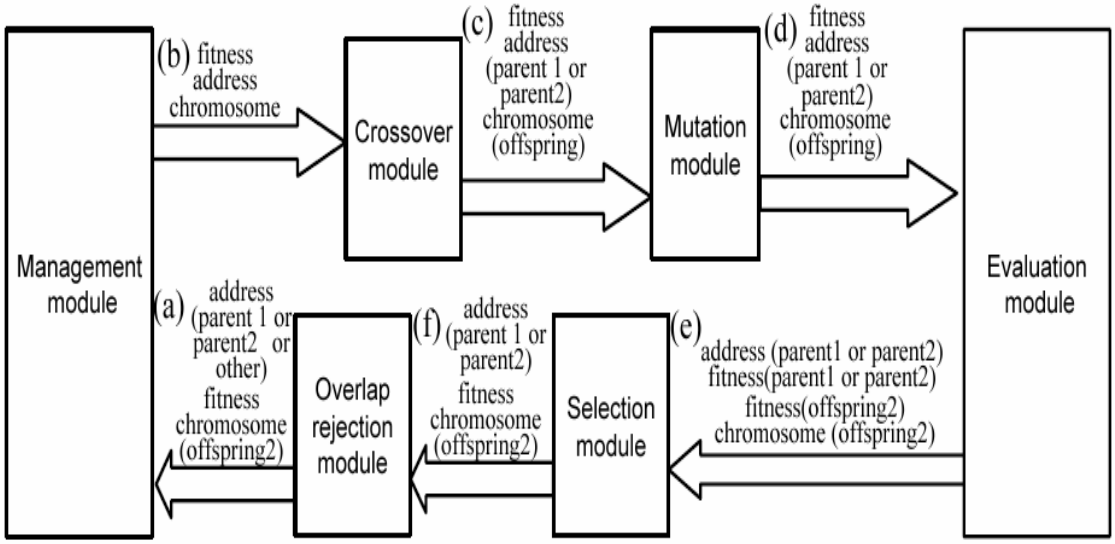

Figure 14: The block diagram of the architecture

This is the block diagram, Figure 14, of the whole architecture consists of the management module, the crossover module, the mutation module, the evaluation module, the selection module and the overlap rejection module. The management

module is the place where we keep the population. It reads individuals from the memory and sends them to the crossover module. Moreover it accepts members from overlap rejection module. In the crossover module we have a register which keeps chromosomes, address and the fitness value from the first parent. It makes crossover to the two parents and generates the new population. It also combines the fitness values of the two parents and sends the fitness value and the address of the dominant. The mutation module mutates the children it takes from the previous stage and sends the offspring 2 and the contents of two parents to the next module. Evaluation module calculates the fitness value of the offspring 2 and sends it to the selection module. The inputs of the selection module are the chromosome and the fitness of the offspring 2, the addresses and the fitnesses of the parents. The last module updates the population controlling for repeats and removes them. Figure 15 introduce the parallel architecture.

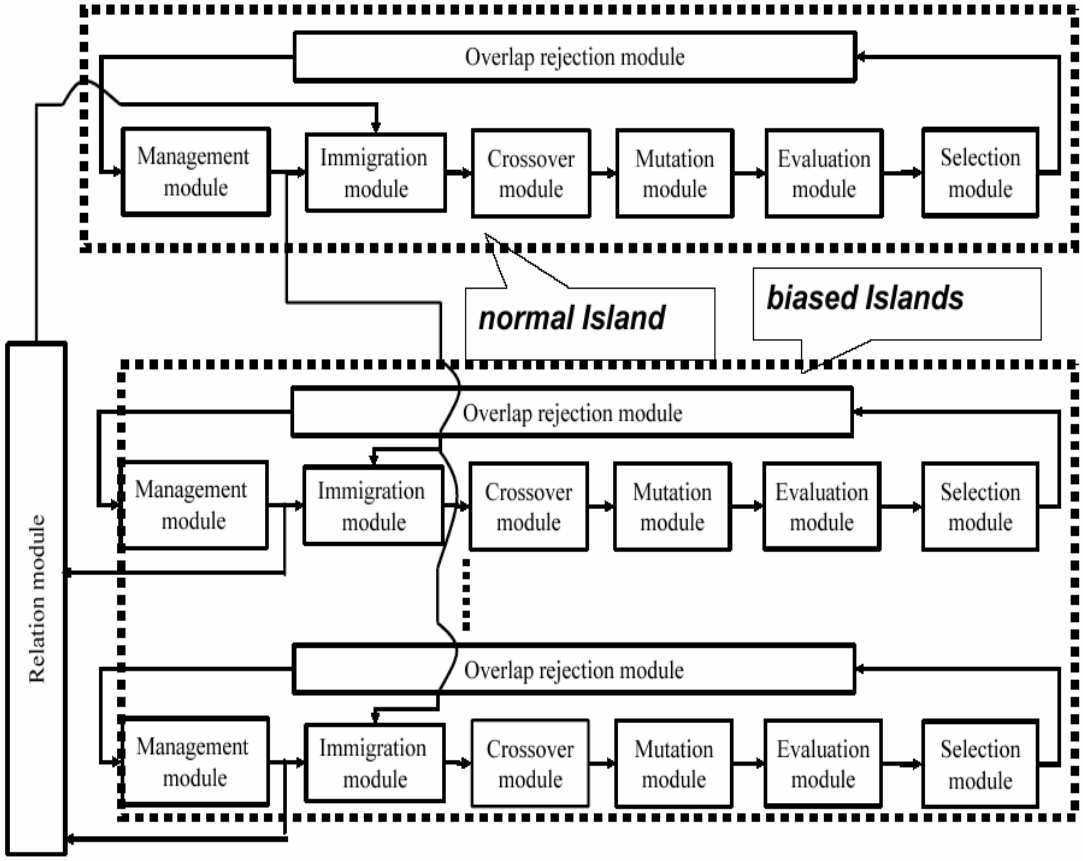

Figure 15: The parallel architecture

Tachibana et al. compared the proposed method with a software version called NSGA II. The selected parameters were: 64 individuals, crossover rate 0.6 and mutation rate 0.02 and they realize that the method they proposed performs better for the FPGA of 100 – 140 MHz than the NSGA II software. The Table 14 below shows the results of the two methods.

Method	Pareto Solutions	Evaluations per Island	Total Evaluations	Processing time (sec)
Normal Method(1)	34.6	1,000,000	1,000,000	0.01
Normal Method(2)	36.1	1,000,000	2,000,000	0.01
Normal Method(4)	39	1,000,000	4,000,000	0.01
Normal Method(6)	39.6	1,000,000	6,000,000	0.01
Biased Method(3)	46.6	1,000,000	3,000,000	0.01
NSGA-II	33.8	320,000	320,000	43.2

Table 14: Comparison of FPGA and NSG

2.2.10. Koza et al

Koza et al. described how the parallelism of the Xilinx XC6216 FPGA can accelerate the fitness function in a genetic algorithm. The most time-consuming task is the fitness evaluation of each individual. All the individuals are implemented in hardware. The Xilinx XC6216 FPGA contains a 64 × 64 two dimensional array. Moreover, it consists of 4096 logical cells and each of these cells contains multiplexers and a flip-flop. For the routing are used 24 bits and the combination of these 24 bits doesn't create conflicts and also there are located in the address space of the host computer. Sorting networks are algorithms for sorting elements using comparisons and exchanges if this is needed. In the Figure 16 below we have 4 elements A_1, A_2, A_3 and A_4. We compare A_1 and A_2 with the vertical line shown below and if A_1 is grater than A_2 are exchanged. So with this algorithm we put the grater element at the bottom. Consequently, the algorithm makes 5 steps for the 4 elements of the network [7].

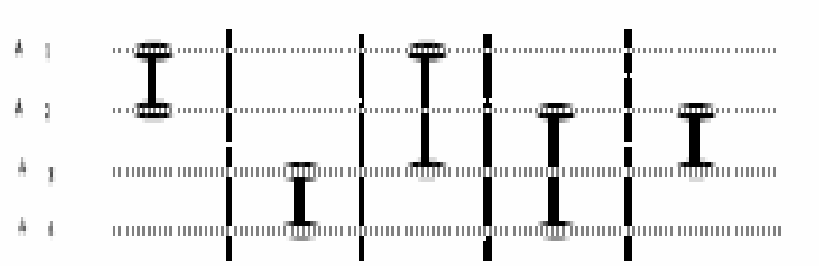

Figure 16: Minimal sorting network

After describing the sorting network above, Koza et al. tried to map the problem onto the chip. All the processes run on the host computer but the fitness evaluation is performing in the Xilinx FPGA because it is the most time-consuming operation of the algorithm. Figure 17 shows the placement on 64 vertical columns and 32 horizontal rows of the XC6216 Xilinx of 8 fields. The A sector is the control sector and B creates the fitness cases. In addition, the fitness cases are sorted in fields C, D and E and are evaluated in areas F and G. Moreover the C area corresponds to a compare-exchange function. The D operates as a forward field from the C area to E. The output of E is controlled from the F module to make sure that everything is good. The 16 bit accumulator G is incremented by 1 if the bits are sorted suitable. The 2 areas C and E represent the candidate sorting network. In C area, each cell in a 16 × 1 vertical column is configured with 3 different ways. One of the 16 cells is configured as a two argument Boolean AND function. One other cell is configured as a two argument Boolean OR function, and the third way of configuration of the other 14 cells is "pass through" cells, which pass their input to the next vertical column. These 3 ways of configurations are also the same in area E. The output of each logic cell is one bit length and it's stored in a flip-flop. Then, the 16 flip-flops in a vertical column are inputs to the next vertical column. When the process starts all the 16 × 80 flip-flops (C and E area) are initialized to zero. So the first 87 vectors (80 from C and E area, 7 from F and G area), consists of 16 zeros and the accumulator G doesn't increment. Then the counter B starts counting (past zero flip-flop is enable) from $2^{16} - 1$ and when it reaches $2^{16} - 87$, area A stops the increment of accumulator G. The output from G (fitness raw) is taken from the reporting register H and the done flip-flop is set 1, so the host computer understands

that XC6216 has completed the fitness measurement for one individual. Koza et al run the design at 20 MHz in the XC6216 which is 10 times slower than a serial microprocessor. It is important that all of the operations performed in areas A, B, C, D, E, F, G, H are parallelized in the FPGA. Moreover, Boolean AND numerous and OR operations are performed parallel. The 19 step 8 sorter was evolved on 58 generation with population size = 60,000. In other hand, the 25 step 9 sorter was evolved on 105 generation with population size = 100,000. Finally, both of these sorting networks implemented in FPGA were minimal.

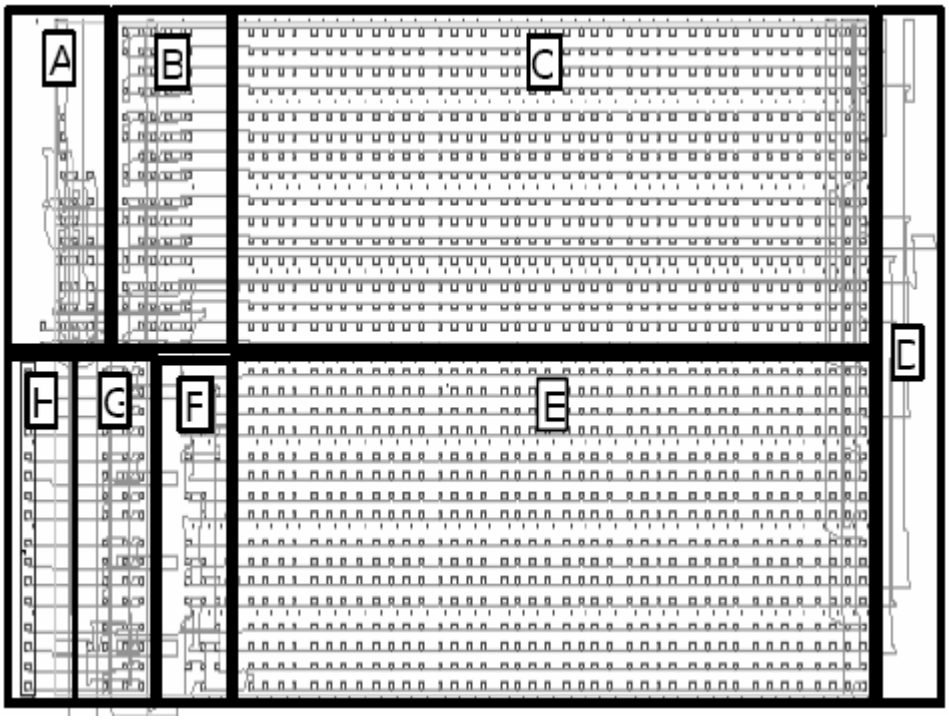

Figure 17: Placement of a 32 × 64 portion of XC6216 FPGA chip

2.2.11. Heywood et al

Heywood et al proposed a register based genetic programming on FPGA. The FPGA computing platform is used for linearly structured genetic programs where the individuals are described as number of pages and page length. Before introducing the instruction format we must talk about register machine. Register machine, as defined by

Von Neumann is computer machine consists of registers which are depended on the operations of CPU. The basic functions are fetching, decoding instruction and manipulating the contents of several registers. The simplest Von Neumann register machines have limited set of simple registers. If the order of instructions is wrong the program stops. We can use four register addressing modes. The 0-, 1-, 2- and the 3-register addressing mode.

We observe that when we use small addresses in registers we benefit of hardware but the code we produce isn't efficient. The operations of a genetic program are 1) the decoding cycle, 2) the calculating of the cost function, 3) the crossover and mutation operations, 4) memory management and 5) stochastic selection. The bottleneck of this system is the memory access. It is assumed that the initial population of individuals is performed by the host computer. The stochastic selection is achieved either using hardware random generator with left shift register or the random numbers are produced off-line. For the computation of the cost function a scalar square error is assumed. In the classic implementation of the crossover operation we face the problem of memory management because code segments with different length size are crossover. So it must be bounded enough memory space for each individual. A lot of jump instructions are used to control the program flow. This implementation requires a lot of clock cycles and the two instances are implemented directly in hardware. So Heywood et al, proposed a different approach for the crossover function. They initialize the individuals by defining the number and the size of program pages. The pages consist of a number of instructions and the crossover operator chooses which pages are going to be swapped between the two parents. Only one page per time is crossover of the two parents. The memory reads programs and copies the contents of the parents to children. In the case of mutation operator random instructions are selected and then performing an EX – OR operation with second random integer. Moreover, the exchange between two instructions of the same individual is performed because the order that the instructions are executed has effect on the efficient of the system. A technique indicating the calculation attempt of the algorithm for the 4 different address formats is the relationship between the generation and the number of

individuals processed. We present the best and the worst result in the Figures 18, 19 below. For the 1-address format the individuals the algorithm processed for 20,000 generations are 750,000. In the other hand, when we use the 3-address format for the same number of generations the genetic algorithm processed 400000 individuals.

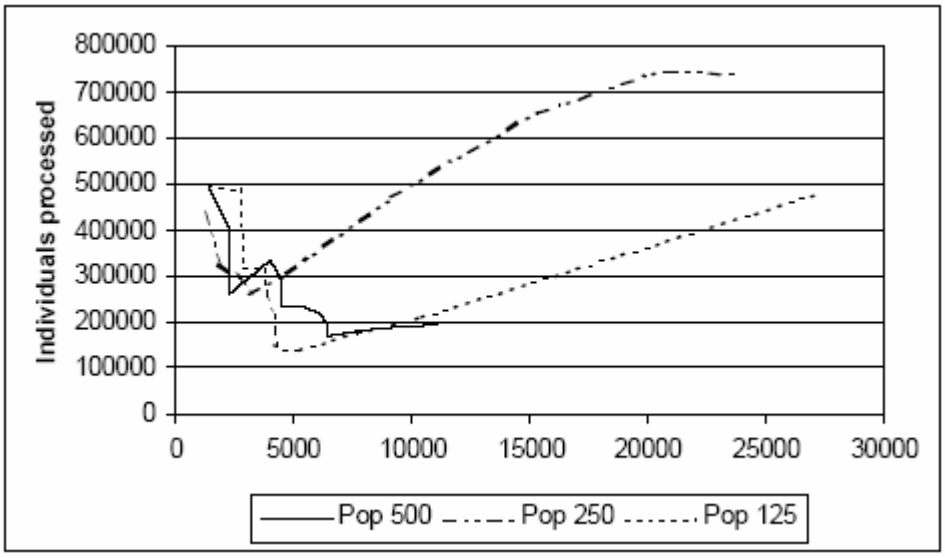

Figure 18: 1-address instruction format

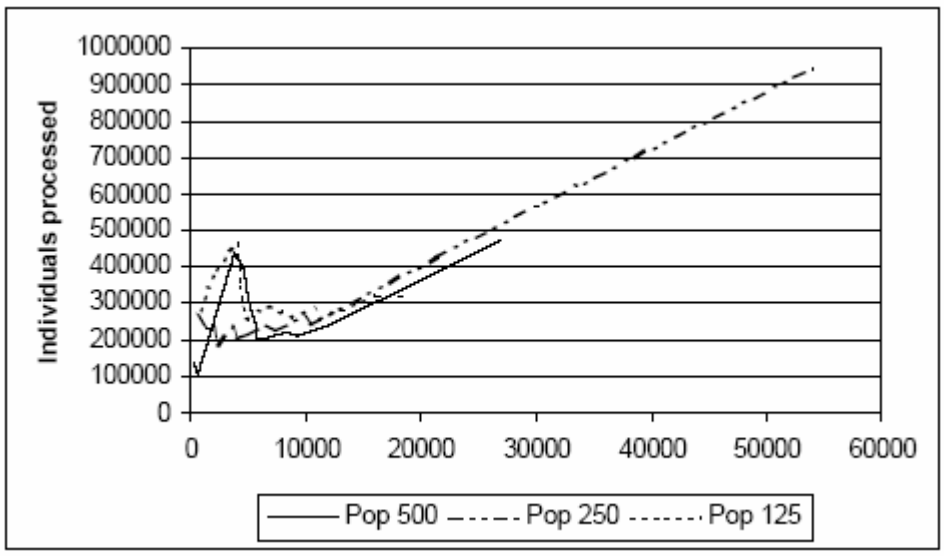

Figure 19: 3-address instruction format

2.2.12. Perkins et al

Perkins et al, presented a self-contained FPGA based implementation of a spatially-structured evolutionary algorithm for signal processing. This algorithm provides speedup because of the computation of individuals in parallel. In addition, we get enough space and we can work efficiently in the FPGA [13]. Finally, we put the entire algorithm in the FPGA and we benefit in speedup due to the minimizing of the time required for the chip - host computer communication. In this paper, Perkins et al, evolved the population of a non linear digital filter, into a simple Virtex FPGA in order to solve a non trivial 1 × dimensional reconstruction signal problem. An important class of reconstructed filters is stack filters. The stack filter is a sliding window non linear filter and the output in each position of the window is determined by applying a positive Boolean function (PBF) on a decomposed threshold. We apply the stack filter on the 1 × dimensional string. The result is a two dimensional wall whose the height is the position value. Each of the one dimensional rows of the wall gives a Boolean string where 'true' represents inside the wall and 'false' out of the wall. Then we apply the PBF. The threshold decomposition part of the stack filter is time – consuming. For 8-bit signal we take 256 different levels and each operation of the filter requires 256 calculations of PBF. Utilizing Chen's technique based on binary search we have PBF calculations equal to bits of the signal. The input values of the filter arrived as parallel bit-streams, most significant bit first (MSB). Firstly, the PBF applied to the MSB and we take the MSB of the output. If there are input bits which have different value from the output bit then the input bit 'locked' in its present value. PBF is applied to other bit- streams producing the output bit and this process repeats. Chen's method results are almost the same as stack filter when the Boolean function is positive. Although, producing Boolean functions and check if they are positive is time - consuming for the genetic algorithm. In this genetic algorithm proposed by Perkins we use a window of 5 elements and the Boolean function defined by a truth table with 32 × 1-bit values. So, we have 2^{32} possible truth tables but 7581 only represent positive Boolean function. The filter used here is an arbitrary Boolean filter and it is stack if the Boolean function is inside the 7581 values and something else otherwise. We call these filters 'stick' filter.

For a 5- element window the arbitrary Boolean function is represented by a 32 element truth table. We use direct genomic representation and the genome for each individual in the genetic algorithm is a binary string giving the truth table for each Boolean function. We have a population of 48 cells 6 × 8 grid on FPGA. Cells are initialized with random truth tables and each cell has an error counter which is initially set to zero. After the initialization, each cell receives a corrupted Sc and in-corrupted signal Su. In each step we apply the stick filter in a window consists of 5 samples of Sc signal and we produce the S_r reconstructed signal. Then we take the difference between S_r and Su and this number added in error E. If the cells have error, the genetic algorithm operates at the breeding mode. Each cell checks its error with 4 neighbor cells at north, east, south and west. When the fittest found, we have uniform crossover between the selected cell and the fittest. If the cell we select is the fittest we don't change anything. After the crossover operation, it is the point mutation on each element in the truth table of each cell with probability Pm. When breeding operations are finished the error counter is set again to zero and we repeat the process. We target this genetic algorithm on Annapolis Microsystems Wildcard. This PCMCIA card contains a Xilinx Virtex 300 part and two independent banks of 256Kbytes SRAM. The results of these implementations are shown in Table 15 below.

Component	Number of CLB Slices	Percentage of Total
48 Cell Array	1904	62
Controller	43	1.5
Wildcard Interfaces	549	18
Total	2496	81.5

Table 15: Results

2.2.13. Tachibana et al

In this section we introduce a general architecture for hardware implementation of genetic algorithm proposed by Tatsuhiro Tachibana. The architecture below consists of 4 modules, the management module, the crossover module, the mutation module and finally the evaluation module. Each chromosome is coded as a string of n bits [14]. The

buses between two successive modules have width of m bits. If each module receives m bits of data and outputs m bits in a clock cycle then n/m clocks are used to process the chromosome. Moreover, all the modules receive and process data in parallel. For eliminating the required memory we assume the minimal generation gap model. We select two individuals (parents) from the population and we perform crossover and mutation so the offspring is generated.

The first module is the management module which stores the population in memory. The address and the fitness function of the parent with the worst fitness value and the chromosome and the fitness value of the offspring are received from the evaluation module. We compare the two fitness values and if offspring's fitness value is higher we send the address and the chromosome of the offspring to the crossover module. In the crossover module we have a register which retains the address and the fitness value of the last individual received from the management module. We apply crossover in this individual and the individual arrived from the management module and we generate the offspring. We compare the fitness values of the parents and we send the worst one and the address in the next stage. The chromosome of the offspring is also sent.

After the crossover module there is the mutation module which mutate the offspring and the resulting offspring is sent to the next module. In addition, we send the fitness value and the address of the worst parent. In the last evaluation module we calculate the fitness value of the resulting offspring and we send the address and the fitness value of the worst parent and the fitness value and the chromosome of the offspring. We have better results with this technique than the software genetic algorithm. With more pipeline levels we obtain lower fitness value and best efficiency.

2.2.14. Lei et al

Lei et al proposed a hardware implementation of genetic algorithm with FPGA. The hardware architecture of the genetic algorithm consists of I/O interface, processing unit and control unit. The operations of the processing unit include the initial

population, the calculation of fitness, selection, crossover and mutation. The control unit operates as control state machine [15].

In this design there are five modules in the processing unit: 1) generation, 2) mutation, 3) crossover, 4) random number generator and 5) selection. The state machine of control unit is used to decide the sequence of these five modules and sends control signals at the processing unit. Processing and control unit are cooperating for the total function of the genetic algorithm. Hardware of the genetic algorithm is controlled by three outside signals. The RUN signal which specifies the global startup, the global RESET signal and the clock CLK signal. The control state machine of control unit generates five output control signals and is controlled by four signals o1-o4 from processing unit signal. The modules of processing units are controlled by the control state machine and are operating in two states, active and sleep. Below we explain the functions of these five modules and the control state machine. Random number generator is implemented as LSFR register and the numbers it generates have size n=12 bits. A random pseudo number is outputted from two ports, dout1 (8bits) and dout2 (3bits) by a control signal which is generated from the state machine of control unit. Storage module is the first step of the genetic operator running in two modes, generation and storage. Generates the initial population and calculates the fitness values of individuals when is working in generation mode. Reads the new population from population output ports of mutation module and stores the new population into registers. Moreover, calculates the fitness values, when is working on storage mode. A divider is applied for the evaluation of the fitness values.

After the storage module there is the selection stage which reads the population and the fitness values from storage module and selects the individuals depending on their fitness values. The roulette wheel algorithm is used where all the fitness values are used for the evaluation of the probability values. Given a random data, the roulette runs, and the pointer points the area of fitness value. The individual corresponding to this area is selected. This is done four times and the new population consists of the selected individuals which are stored into an array called *new_mem*. The individual that has the highest value has more probabilities to be chosen. In the crossover module we

read the population and three random data and we target them into registers. We use single point crossover and this operation is finished randomly. The mutation module is the final step of the genetic algorithm. It keeps the diversity and avoids converging local results so early. The probability is small (1 / 512) and the point of mutation is selected randomly. Finally, the control state machine is the core of control unit. It can achieve high speed transformations but small setup time in each I/O. It consists of six states: idle, birth, storage, selection, crossover and mutation.

The genetic algorithm described above was implemented with VHDL in a Xilinx xc2s100 FPGA. At 20 MHz clock frequency it took 0.15 seconds for 1,000 generations and it was 1000 times faster than the software implementation with 200MHz.

2.2.15. So et al

So et al. presented a four-step genetic algorithm (4GS) implemented in FPGA. The cost of calculation is similar to three step genetic algorithm. 4GS can be applied in video encoding hardware [16].

At the initial population each chromosome represents a motion vector. The length of the chromosome is 4 and the maximum displacement l is 7 pixels. Continuing there is the evaluation stage where the fitness value of each chromosome is calculated. In this design four chromosomes of the total 16 are selected for reproduction. In the reproduction stage n chromosomes are selected for reproduction from N and are transferred at the mutation pool. This method is similar with the roulette wheel. The mutation pool consists of N new reproduced chromosomes. Each of these mutated form a pair of operators. The size of mutation depends on each generation. For the three first generations the size is two and for the fourth is one. We must mention that we have 8 different pairs of operations. Motion vector is the chromosome with the lowest fitness value in the population. N chromosomes are selected of the 2N in each generation and these with the smaller fitness value are selected for the next generation. The steps from evaluation stage till selection are repeated for 4 generations and the

search algorithm is terminated when we have 4 generations or the fitness function becomes zero.

Describing the hardware of the four-step genetic algorithm, it is worth noticing that is separated in two parts, the interface module and the core module. The first is responsible for the memory and the core and the last one searches the motion vector. The external frame memory consists of several memory blocks. In this implementation they choose block size 16 X 16. The total number of memory blocks is 16 and the width of memory bank is 512. Moreover, the size of each memory bank is 16k. So, used a 16 bit random number and the fitness function is evaluated by an array. The array is calculating the difference between pixels and partial fitness function. The difference unit consists of two subtractors and a multiplexer. We have a one dimensional array with 3X3 size. The evaluation module takes the fitness function (2 X 16 – 1) cycles later and examines which of the chromosomes are matching for this environment. The population module controls the whole searching procedure. In each generation N chromosomes are produced but in the end n survive. The population unit initializes the searching algorithm in each block. A randomly selected chromosome parent will be mutated from this unit. The evaluation unit evaluates new chromosome every M cycles and the population unit produces new chromosome every *M* cycles. When the searching procedure is finished the population unit updates the value of the motion vector and the fitness function. In each generation are produced chromosomes every 16 cycles, so there are 16 new chromosomes and only 4 survive.

For his design So used two Xilinx 4025 chips with 60% CLB utilization of the total number of 1024 CLB's each. For searching one motion vector, the 4GS needs 1,152 clock cycles. With minimum clock frequency at 11,4Mhz the proposed design can handle video sequence with 352 X 240 size and frame rate 30Hz. In the case of MPEG-2 the frame size is 750 X 576 with block size 16 X 16 at 30 Hz rate and the maximum clock period is 17,86nsec.

2.2.16. Graham et al

Graham et al described a hardware genetic algorithm for the traveling salesman problem on SPLASH 2. SPLASH 2 parallel genetic algorithm, is a hardware based genetic algorithm searching for an optimal solution in traveling salesman problems. This family of problems is searching the shortest path between n cities visiting each city once and returning at the initial city. Each possible solution of the population consists of an ordered list describing the sequence in which each city is visited, called tour. The fitness of each tour is related with its length. Two tours are selected for crossover and a random cut point is selected, so the tours are cut at this point. The head of tour A becomes head in the offspring A and the head of tour B, head in offspring B. The tail of offspring A is formed by taking the cities form tour B not contained in the head of tour A. The tail of offspring B is formed in a similar way. Mutation is performing on the selected tours by reversing the order of cities visited within a sub-tour contained within the original tour. The endpoints of sub-tour are selected randomly [17].

SPLASH 2 is a reconfigurable computer consists of an interface board and a collection of processor array boards. It is programmed with VHDL. The basic computational module is a processor consisting of 4 FPGAs Xilinx 4010s and their memories. These four FPGAs are forming a pipeline. During the execution, memories store the current generation, the new generation, the fitness values of tours, the array of distances between the cities and some other operation parameters. The initial data are supplied by the host address. The function of each FPGA is presented below.

FPGA 1 uses biased selection, choosing pairs of tours from the memory. This is achieved with hardware-pipelined roulette wheel algorithm. Initially, one random number is produced, called the target. Tour fitness values are sequentially accumulated until the target reaches a specific value. The tour that causes overflow is the one that is selected. Tours with highest fitness values are preferred than tours with lowest values. As soon as two pairs of tours are selected, their index numbers transferred into FPGA 2 through a pipeline path. FPGA 2 has two choices: it can copy the tours at the right without changing anything or it can combine them through crossover and send the new

offspring at the right. This decision is taken from a random number generator on the chip and the crossover probabilities are 10% to 60%. FPGA 3 calculates the fitness values of the tours produced from crossover. In addition, it randomly selects tours for mutation and sends the tour pairs and their fitness values to FPGA 4. FPGA 4 writes the new population into memory determining the best and worst tours of the current population. The above process repeated until the population size becomes equal to the original size of FPGA 4. The pipeline, copies the new population and the fitness values back at memories of FPGA 1 and FPGA 2 and this terminates the first generation.

The implementation proposed by Graham et al. requires 3500 code lines in VHDL. Assuming maximum clock frequency at 11 MHz the CLB utilization found from 37% to 60% for all FPGAs. The performance of a software implementation at 125 MHz HPPA-RISC Workstation is compared with SPGA in Table 16 below.

Number of Cities	Population Size	Crossover Probability	Mutation Probability	Average Execution Time (sec) Hardware	Average Execution Time (sec) Software	Software/ Hardware
24	128	10	10	4.38	43.7	9.97
24	256	10	10	11.23	118.7	10.57
120	256	60	10	295	1999.9	6.78

Table 16: Comparison of hardware and software execution times

Two parallel implementations proposed by Graham, the trivially parallel model and the island model. The implementation described above utilizes 4 FPGAs with simple SPLASH 2 board. The remaining 30 FPGAs and the memories at SPLASH 2 two-board are idle. In fact, there is no need of SPLASH 2 for the basic implementation. However, given a SPLASH 2 two-board, it is an extension for running 7 additional copies of the algorithm and isn't required additional hardware design. The copies of the algorithm don't interact each other and the time we spent is the initialization time of the memories. An 8-fold increase in search rate is possible with the trivially parallel model. The other approach of parallelization of the algorithm is the island model.

Several searches are performing in the periodically migration of the solutions between the islands. During the migration, each island broadcasts a subset of tours to other islands via crossover and the islands that receive these tours replace the old with the new ones. In contrast with the trivially parallel model, it is needed modifications of the SPGA. With a comparison between the three proposed methods, the 8-processor trivially method searches faster than software from 54 to 85 times. However, speedups don't indicate better solution in the terms of quality but a greater number of evaluations. So, if our goal is to find the best solution, the 8-processor parallel version is 4% better than single and 4-processor island model is better by about 6%. If we aim for a quick solution than island model is the best of all. At 500 million cycles 4-processor island model finds a solution when 990 million cycles is needed for 8-processor trivially parallel version and 1.7 billion cycles for the single processor.

2.2.17. Glette et al

An online evolution for a high-speed image recognition system implemented on a Virtex-II Pro FPGA has been proposed by Glette et al [18]. The architecture is implemented as circuit and the behavior and the configurations of it are controlled by configurations registers. The system consists of three main parts: 1) the classification module, 2) the evaluation module and 3) the CPU. This is shown below in Figure 20.

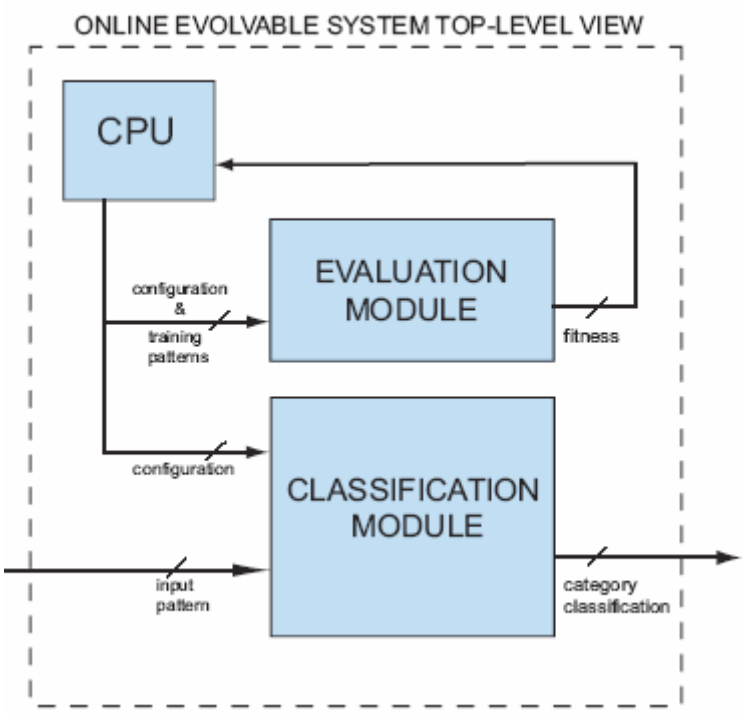

Figure 20: Top-Level view of online evolvable system

The classification module operates uniquely except the reconfiguration accomplished by the CPU. The evaluation module cooperates with the CPU for the evolution of new configurations and accepts a configuration bit-string called genome computing its fitness value. This information is used by the CPU to run the genetic algorithm. The classifier system consists of K category detection modules (CDMs), one for each category. The input data to be classified is presented in each CDM the same time via common input bus. The CDM with maximum output value is localized from maximum detector and the number of this category will be the output of the system. Each CDM consists of M rules or functional units (FU) and each FU row has N FU's. The inputs of the circuit pass on the inputs of each FU. The 1-bit outputs of FU's are getting into N-input AND gate sequentially. This means that all of the outputs must be 1 to activate the rule. In addition, the outputs of the gates are connected on an input counter which counts the number of activated FU rows. FU's are the reconfigurable elements of this architecture. Their behavior is controlled by configuration lines which are connected to configuration registers. Each FU has all input bits on the system

available at its inputs, but 1 data element (1 byte) is selected, depending on the configuration lines. This data is then get into the appropriate functions whom number and type are varied. The unit is configured by a constant number C, so this value and the input data element are used by the function for the computation of the output of the FU. In this implementation Glette et al. utilize 8 FU's.

The fitness function of the face image recognition is based on the ability of the system to recognize the correct person from several faces. These faces are taken from a database that has 400 images, that is to say 10 images from 40 different persons. The image resolution is 92 X 112 pixels, 8-bit gray scale. For reducing noise and inputs, Glette et al. down-sample the images to 8 X 8 pixels, 8-bit gray scale. The input pattern is 64 pixels and types of functions in FU's are greater than and lees than or equal. The constant value is 8 bits. For the FU implementation they used a scheme where 1-bit exists in the FU in a single time. The 64 pixels are presented sequentially, one in each clock cycle with its address. Functional unit check the address of the input pixel for matching with the address of pixel exists in the configuration register and if they are the same, the value of the output of FU is stored into a memory element. This method requires 64 clock cycles, before FU selects its input. In the evaluation module, is computed the fitness value for a FU row which is the phenotype for an individual in the evolution. After the configuration of the FU via the register, FU row must be fed with all the faces of all the categories. There are 360 images (9 face images for 40 categories) and they are loaded before the process in the image memory Xilinx BRAM which is located at the evaluation module. As soon as the configuration register is written, the process starts. The address generator produces addresses for images. Control logic cells urge fitness counter to sample the output of FU row and this happens after the 64 pixels have been cycled. Then an interrupt is sent to the CPU and the fitness value is read. Moreover, fitness value can be duplicated for extended parallelization. In this design there are 8 FU rows in parallel for the computation 8 individuals.

The genetic algorithm follows the pattern of simple GA and the algorithm is running at Power PC 405 core in Xilinx Virtex-II Pro FPGA. 15-bits for each FU are

required for the encoding in genome, 6 for pixel the address, 1 for the function and 8 for the constant. So, 120-bits (15 X 8 FU rows) are the total number of bits for 1 FU row. The fitness evaluation is done by the hardware evolution module and other individuals are produced by the CPU. The evaluation module sends an interrupt to the CPU that finishes its work and takes the next individual. Online evolvable hardware system run the fitness evaluation in a Virtex-II Pro xc2vp30 FPGA at 131 MHz with 1393 slices of total 13696 (10%), and the other genetic operations run on 300Mhz Power PC. The population size was 16. These results are presented in Table 17.

Resource	Used	Available	Percent
Slices	1,393	13,696	10
Slice Flip Flops	1,419	27,392	5
4 Input LUTS	1,571	27,392	5
18 Kb BRAMS	17	136	12

Table 17: Device utilization for the 8 row evaluation module

In conclusion, we compare the EHW with a software version Intel Xeon 5160 workstation which had from 10 to 30 times the clock speed of EHW. The speedup over software was found 1.01 totally for 1000 generations. It is worth noticing that Xeon spends a lot of time for fitness evaluation because it is implemented in software. In EHW the fitness evaluation is implemented in hardware and this indicates the significant differences in times. The results are presented in Table 18.

	EHW	Xeon	Xeon/EHW
GA Clock Frequency (MHz)	300	3,000	10
Fitness Clock Frequency(MHz)	100	3,000	30
GA Time (ms)	926	9	0.01
Fit. Eval. Time (ms)	623	1,323	2.12
Total Time (ms)	1,313	1,323	1.01

Table 18: Speed comparison between Xeon and EHW

2.2.18. Shackleford et al

A high - performance pipelined FPGA genetic algorithm machine proposed by Shackleford et al. The population of n_p chromosomes with the fitness value stored in a 1-dimensional array called population. Each entry in the array consists of n_d data bits and n_f fitness bits. The function random (*i*) returns a random number from 0 to (*i-1*) for each $i > 0$. Fitness (c_{data}) function evaluates how fit is the solution and returns an integer from 0 to (2^{n_f-1}). The best solution is represented by the greater value of fitness values. The crossover (*cut_prob, parent1, parent2*) function results the child which is a combination of parents from cut in random points. The crossover probability could be 0 (for single point crossover), 0 – 0.5 (for multi point crossover) and 0.5 (for uniform crossover). The mutation (*mutation_prob,* c_{data}) function changes the data of the selected chromosome. The mutation probability takes values from 0 to 0.5 for single, multi and randomization mutation. A mutation probability of 1 inverts the selected for mutation chromosome. The algorithm is terminated when the evolutionary stasis function returns a true value [19].

Initially, random chromosomes produced with their fitness value and stored into the population array. The old parent 1 becomes the new parent 2 and the new parent 1 is selected randomly from the population array. Only 1 memory access cycle required for each crossover operation. The basic idea of this steady state genetic algorithm is the survival of the fittest child and parents with worst fitness values that are randomly selected for crossover and mutation, are candidates for replacing by the survival child. There are 2 variables, "worst_fittness" and "worst_address" which store the fitness and the address of the worst parent. Two parents are selected and the child chromosome is created by the crossover and mutation operations. Its survival depends on its fitness value which if is better than less fitter parent, the child chromosome is stored in the population array at the worst parent's address. So, the total fitness of the population is incremented until a terminate condition is met. The population array because of a steady state GA, is implemented like simple memory which results in chip are savings. Parents are selected randomly and we don't need a circuit to choose the best one. Crossover and mutations operations performed each clock cycle creating a child

chromosome in this time period. In addition, crossover and mutation architectures are based on shifting template register which eliminates long wires.

The FPGA-based GA machine consists of 6 six-stage pipeline. The first three stages are dedicated for parent selection, the fourth stage performs crossover and mutation, fifth stage is responsible for the fitness evaluation and the last stage is the survival stage. Random number generator (RNG) is implemented as cellular automata and in every clock cycle several random numbers are produced. During the population memory access cycle, memory can either be read or written. This is decided by a comparator at sixth stage and when comparator's output is zero reads the address from RNG and when it is one writes the address from the last stage of the pipeline. First parent stage stores the first parent with his address and his fitness value. Loading registers from this stage are cut off and the reason is to prevent the survival child, written to memory, to affect the evolution. At the beginning of second parent stage, old parent 1 becomes new parent 2. So, even if the memory has one single read port, a new pair of parents is presented for crossover in every cycle. Crossover and mutation are performing at the same time and the worst parent is selected for replacing. At the fifth stage the fitness value of child chromosome is evaluated and if fitness function contains long logic paths a pipeline is required. After fitness evaluation, the new child exists on register child and is compared with worst parent stored at the least-fit register. If is better, it replaces the worst parent in the population memory and the worst parent is discarded.

Datapath represents a significant portion of GA machine, so it should be implemented efficiently. When a surviving child is written to the population memory, it should be prevented from re-entering the pipeline. So, there are hold controls on the parent registers which are active when we are at population memory write cycle. Each bit requires a two input multiplexer to select the parents. The control of multiplexers is achieved by sending a crossover template to all multiplexers via shift register. This requires 1 flip-flop for each slice and permits several values for cut-points between 0,05 to 0,5. The crossover pattern is created probabilistically by a comparator connected to threshold register and a RNG which generates a random number. When the random

number is less than threshold, the comparator output is true causing the change of the state of flip-flop and the selection pattern applied in the template shift register. The information of mutation is transmitted to the data bits serially through shift registers. In order to reduce the probability of mutual influence we have two bit-streams which travel in a different direction. The occurrence of mutation for a bit is defined as two ones appearing simultaneously in the same position in each shift register. This incident is realized by a two input AND gate which causes the XOR gate to invert the bit at the position from the crossover multiplexer. These two gates are implemented as 3 input LUTs. The two bit-streams are constructed as the template of crossover but the flip-flop isn't connected at the output of comparator. Mutation probabilities take values from 0.01 to 0.05. The result from crossover and mutation is kept at child register which is connected to both the fitness function and final stage of pipeline which sends it to data input register in population memory. We should notice that every stage from the pipeline has the same processing time. Figure 21 shows the FPGA-based GA machine. Shackleford considers two problems for his proposed FPGA machine, the set covering and the protein folding problems. For the first one, he utilized Aptix AXB-MP3 FPCD device which consists of 6 FPGAs, 3 for GA pipeline and 3 for fitness evaluation. Each FPGA is ALtera EPF81188 chip and has 1,008 logic elements. For population size of 256 and a maximum clock frequency of 1 MHz his implementation was 2200 times faster than a software version on 100 MHz workstation. For the protein folding problem he used 70-bit chromosomes and a Xilinx XCV300 FPGA. Pc interface, GA pipeline and fitness function uses 2,000 from possible 6,144 LUTs. At 66 MHz clock frequency he found that his implementation is 322 times faster than a 366Mhz Pentium II. To achieve better speedup over software (9,600 times software), he proposed a larger FPGA with 64K LUTs and 30 parallel fitness functions.

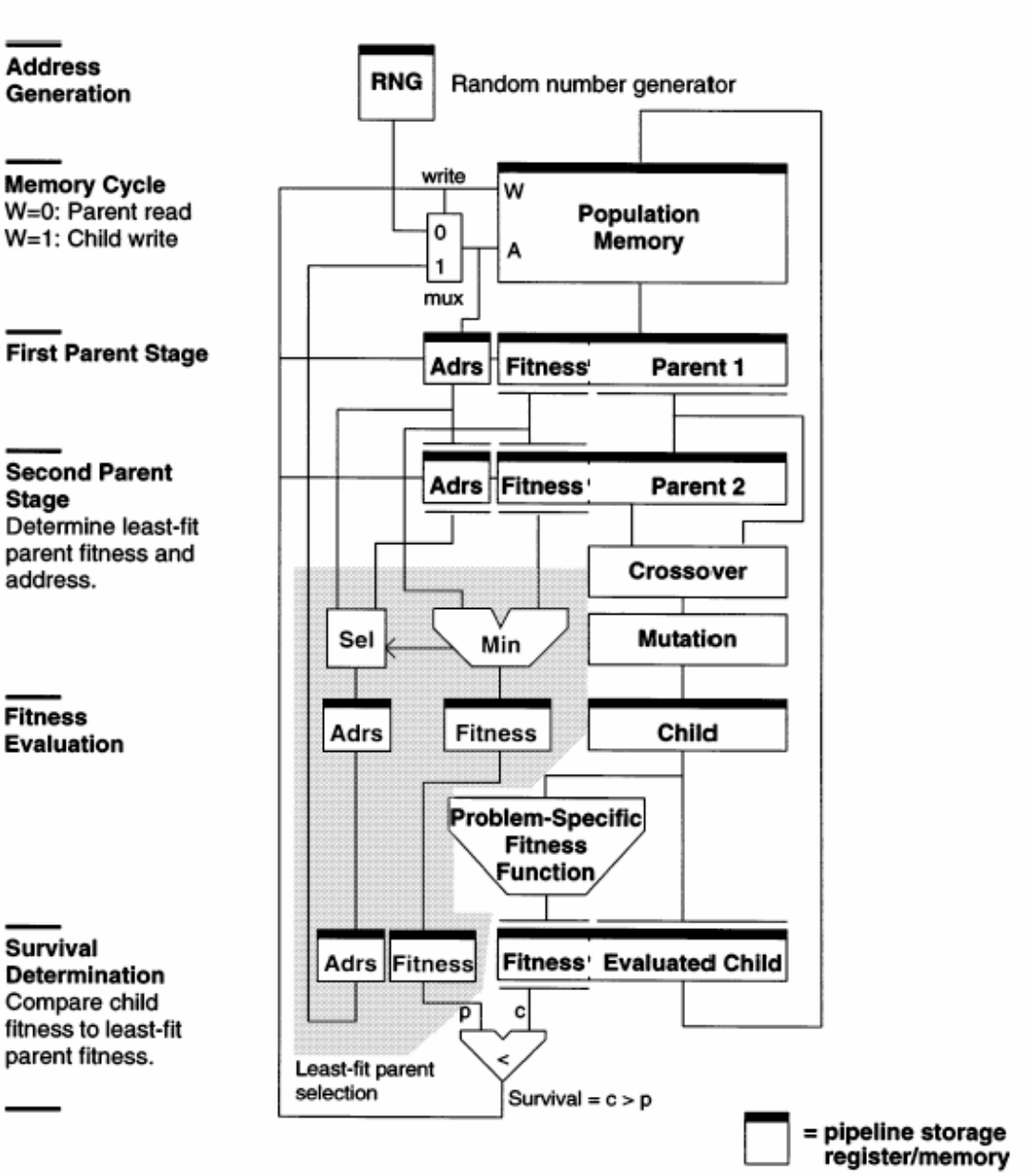

Figure 21: FPGA-based GA machine

2.3. Evaluation and comparison of hardware implementations

This section is devoted to the analysis of the hardware implementation of the genetic algorithms according to criteria such as hardware complexity in terms of configurable logic blocks or slices, execution time and speedup over software.

2.3.1. Hardware complexity

The first comparison criterion between the genetic algorithms presented in this section is hardware complexity which concerns the number of configurable logic blocks or slices needed for the design. Koonar et al [3], utilized a Virtex XCV50E device and the synthesis report shown that 334 slices out of 768 (43%) were used. Moreover, 167 configurable logic blocks was enabled. Tang et al [4], used two FPGAs, a FLEX 10K50 with 360 logic area blocks and a FLEX 6K with 88 blocks. The architecture proposed by Peter Martin [1], was implemented in the CELOXICA RC1000 FPGA Board for two different problems. For the regression problem he realized 4250 slices out of 19200 (22%) for single fitness evaluation and 6800 slices (35%) for the four parallel fitness evaluation. On the other hand for the XOR problem the results were 4630 slices (24%) for the single version and 7434 slices (38%) for the parallel environment. For the compact genetic algorithm Aporntewan [2], exploited the Xilinx Virtex V1000 FPGA with 813 slices out of 12288 (6%), and Emam [11] utilized a Virtex XCV300 device with 1081 slices. A different approach was proposed by Scott [10], implementing the genetic algorithm in Borg's board which consists of five Xilinx FPGAs and a prototyping area which consists of three Xilinx FPGAs. He used 2 XC4003S with 100 configurable logic blocks (CLBs) each, 2 XC4002S with 64 CLBs each and finally 1 XC4003 with 100 CLBs for Borg's board. For the prototyping area he utilized 3 XC 4005S with 196 CLBs each. Consequently, the total number of CLBs used was 1016. Continuing Mostafa [9], implemented the parallel-pipelined hardware genetic algorithm in three chips. The first was an XC4005 FPGA with 196 CLBs, the second was an XC 25100 chip with 600 CLBs and the last one was a Virtex XCV800 chip with 4,704

CLBs. In addition, Koza [7], test his algorithm into a Xilinx XC6216 board with 4096 cells and Tommiska [8], utilized the Altera Flex 10K50 FPGA with 360 logic area blocks. The space utilization in the Virtex 300 used by Perkins [13], is 2496 CLB slices with the percentage of 81.5%. Lei [15], implemented his genetic algorithm in a Xilinx XC2S100 chip with 600 CLBs. The four-step genetic search algorithm proposed by So [16] was utilizing two Xilinx 4025 chips with 1024 CLBs each (60%). SPLASH 2 parallel genetic algorithm proposed by Graham [17], was implemented in four FPGAs Xilinx 4010s with 400 CLBs each, and the utilization ranged from 37% to 60%. The fitness evaluation of the genetic algorithm implemented in a Virtex-II Pro XC 2VP30 FPGA at 131 MHz with 1393 slices of total 13696 (10%) for 8 FU row evaluation module, proposed by Glette [18]. Finally, the FPGA-based GA machine proposed by Shackleford [19], was tested for two different problems. For the set covering problem he utilized the Aptix AXB-MP3 FPCD with 6 FPGAs Altera EPF81188 with 1,008 logic elements each. The protein folding problem was applied on xilinx XCV300 FPGA with 2,000 of possible 6144 LUTs.

2.3.2. Evaluation time

The second criterion of comparison is the evaluation time of each design (speed). Koonar [3], assuming clock cycle of 50 MHz, simulates his design with three different benchmarks for different generations count and different population size. The hardware and software times are shown in Table 19. Tang [4], realized that the time is needed to initialize the population of 256 and 60 generations with the single FGA is 500μsec. For the implementation of the genetic algorithm using Handel-c, Peter Martin [1] runs two tests. In the regression problem, the calculation of single fitness evaluation with Handel-c took 351,178 cycles and the 4 parallel took 188,857 cycles. For the XOR problem the computation of single fitness evaluation took 715506 cycles and the 4 parallel took 384,862 cycles. Aporntewan [2] proves that only 0.15 sec execution time is needed for the FPGA to run the compact genetic algorithm for 256 population size. Only 25.124μsec is the time that the genetic algorithm proposed by Emam [11], needs

for 360 generations and population size 80. Tommiska [8] ascertains that the genetic algorithm requires 160ns for processing time and Perkins [13] genetic algorithm runs in 1.65msec for 50 generations. For 20 MHz clock frequency only 0.15 seconds were needed for 1000 generations in the genetic algorithm proposed by Lei [15]. Searching one motion vector, 4GS took 1152 clock cycles in the design by So [16]. In the case of MPEG-2 the frame size is 750 x 576 with block size 16 x 16 at 30 Hz rate and the maximum clock period is 17,86nsec. The basic algorithm implementation described by Graham [17], took 4.38sec for 24 cities and population size 128 and 295sec for 120 cities and population size 256 as presented in Table 16. He also realized that 500 million cycles are needed for the 4-processor island model to find a quick solution which is faster from the 8-processor trivially parallel method (990 million cycles) and the single processor version (1.7 billion cycles). The total time of the EHW proposed by Kyrre Glette [18], was found 1313 ms for 1000 generations. From this time, 623 ms was for the fitness evaluation in the Virtex-II XC2VP30 device at 131 MHz and 926 ms for the other genetic operators implemented on the Power PC at 300 MHz.

2.3.3. Speedup over software

Speedup over software is the last criterion of comparison between the genetic algorithms in FPGAs. It is worth noticing that speedup= cycles in (software) / cycles in (FPGA), as mentioned in Peter Martin [1] implementation. To start with, Koonar [3], assuming a clock cycle of 50 MHz he finds out a speedup over software is almost 50 and with max clock frequency of 120 MHz the improvement in speed is 100 times the SUN ULTRA10 440 MHz processor system. Tang [4], realized that speedup over software is 10 for 2.4 GHz Pentium 4. The speedup between Handel-c single fitness evaluation with clock frequency 25Mhz, and the Power Pc 200Mhz for the regression problem was 47, in Peter Martin's [1] implementation. In the same problem the 4-parallel Handel-c fitness evaluation at 19 MHz was 88 times faster than Power PC at 200 MHz. For the XOR problem, speedup between Handel-c single fitness evaluation with clock frequency 22 MHz and the Power Pc 200 MHz was 38 and 4 parallel

Handel-c fitness evaluation at 18 MHz was 72 times faster than Power PC at 200 MHz. In the compact genetic algorithm presented by Aporntewan [2], speedup over 200Mhz Ultra SParc 2 was 1000. Scott [10], found speedup over software in a range between 12 and 18 for several fitness functions. In Tommiska [8] implementation, the speedup over 120 MHz Pentium processor was 212. The same algorithm was also run for HP C110 workstation (SPECint92 _167) with HPUX operating system and speedup over this software is 275. Perkins [13], in his design with 20MHz clock rates finds speedup over software almost 1090. In addition, Tachibana [6], compared his proposed multi-objective genetic algorithm to a software version called NSGA II in a Pentium 2.4Ghz and realized that speedup over software is 4320. Lei [15] found that FPGA implementation at 20 MHz was 1000 times faster than software version at 200Mhz. Graham [17] assuming maximum clock frequency of 11 MHz found, speedups over software from 6.78 to 10.57 for the basic implementation of SPGA. The software implementation was on 125 MHz HPPA-RISC Workstation. The 8-processor trivially parallel method searches faster than software by factors 54 to 85. Comparing the three methods for the best solution in terms of quality, 8-processor trivially parallel version finds a solution 4% better than single processor and 4-processor island model is better by about 6% for a long execution of 3.5 billion cycles. For a quick solution, 4-processor island model is 1.98 times faster than 8-processor trivially parallel method and 3.4 times faster from the single implementation. The speedup between EHW (131Mhz for fitness evaluation and 300Mhz for other genetic operators) and Xeon Intel 5160 workstation at 3000Mhz was totally 1,01 for 1000 generations as described by Glette [18]. For the fitness evaluation speedup over software was 2.12 because of the FPGA implementation but for other GA operators speedup found 0.012. In Conclusion, Shackleford [19] for the set covering problem with 1 MHz maximum clock frequency, and population size 256, realized that it was 2200 times faster than a workstation of 100 MHz. For the protein folding problem, the speedup between XCV300 with clock frequency 66 MHz and a Pentium II at 366 MHz was 320. In Shackleford's design it was also proposed that with a larger FPGA, with 64K LUTs and 30 parallel fitness functions, it can be achieved a speedup over software equals to 9600.

Genetic Algorithms in FPGAs	Selection	R.N.G	Crossover	Population size (m)	Number of Generations (g)	Maximum frequency in MHz	CLBs or slices	Speedup over Software	Evaluation time (ms)	Device
Koonar et al	Tournament	LFSR	Uniform	20	20	123	334 slices of totally 768 (43%)	100	1.63	Virtex XCV50E
Tang et al.	Roulette wheel	Cellular automata	Uniform	256	60	N/A	448 total logic blocks	10	0.5	AlteraFLEX 6000 and FLEX 10000
Aporntewan et al.	Tournament	Cellular automata	Uniform	256	1,000,000	23	813 slices of 12288 (6%)	1,000	150	Xilinx virtex V1000FG680
Tommiska et al.	Round Robin	LFSR	1-point	32	1	12.5	25,500 usable gates	212	0.002	AlteraFlex10K
Emam et al.	N/A	N/A	N/A	80	360	58	1,081 slices	N/A	0.025	Uni DAC PCI Board(DSP and Virtex XCV300PQ

										240)
Scott et al	Roulette wheel	Cellular automata	1-point	32	20	2	1016 total CLBs	18	23	BORG Board
Mostafa et al	N/A	Cellular automata	N/A	50/500/300	N/A	N/A	196/600/4,704	N/A	N/A	XC4005/ XC2S100/ XCV800
Martin	Tournament	LFSR	N/A	16	511	25	4250 slices of 19200 (22%)	47	14	Celoxica RC1000 FPGA (XCV2000e)
Tachibana et al.	Normal or biased	N/A	Half uniform	64	1,000,000	100	N/A	4,320	10	FPGA 100-140MHz
Koza et al.	N/A	N/A	N/A	60,000/100,000	58/105	20	4,096 total cells	N/A	N/A	Xilinx XC6216
Heywood et al.	Tournament	LFSR	N/A	125/250/500	N/A	N/A	N/A	N/A	N/A	N/A
Perkins et al.	N/A	Cellular automata	Uniform crossover	48	50	20	2496 CLBs	1,090	1.65	Annapolis Wildcard (Virtex 300)
Lei et al.	Roulette Wheel	LFSR	1-point	N/A	1,000	20	600 CLBs	1,000	150	Xilinx

										XC2S100
So et al.	Weighted Roulette wheel	Cellular automata	N/A	16	N/A	11.4	1,228 CLBs	N/A	N/A	Two Xilinx 4025
Graham et al.	Roulette Wheel	N/A	1-point	128, 256	24/120	11	800 CLBs	11/7	11,230/295,000	SPLASH 2
Glette et al.	N/A	N/A	N/A	16	1,000	131	1,396 of 13,696 (10%)	1.01	1,313	Virtex-II Pro XC2VP30
Shackleford et al.	N/A	Cellular automata	1- point	256	N/A	1	6,048 logic elements	2,200	N/A	Aptix AXB-MP3/Xilinx XCV300

Table 19: Principal Characteristics of Genetic Algorithms in FPGAs

2.4. Conclusions

- In this chapter we compare genetic algorithms which are implemented in FPGA and we realize that they generally run more efficiently in hardware than software. This is because GA's can benefited from hardware techniques like pipeline and parallelization.
- As seen in Table 19 parameters such as population size, crossover and mutation probabilities vary from algorithm to algorithm, so a comparison between them is relative.
- Each genetic algorithm has been implemented differently regarding the basic operations such as selection, crossover, mutation and fitness function and they have been targeted into different FPGAs boards.
- In many of the genetic algorithms described above the compilation and configuration times are not always exposed clearly but speedups achieved by hardware implementations to software versions are significant.

From Table 19 it is clearly observed that the least complexity genetic algorithm in terms of CLBs is Mostafa et al implementation which utilized only 196 CLBs on a Xilinx XC4005 FPGA. The most complex implementation is Shackleford genetic algorithm which was implemented into an Aptix AXB-MP3 board with 6,048 logic elements. From all the implementations described above the faster one is Emam's et al implementation. For populations size equals to 80 and 360 generations the genetic algorithm run for 0.025 msec on a Uni DAC PCI Board (DSP and Virtex XCV300PQ240). The multi-objective genetic algorithm of Tachibana et al and the compact genetic algorithm of Aporntewan et al were run very fast but they consist two different categories of genetic algorithms than other simple genetic algorithms implementations. The most time-consuming hardware implementation was Graham et al genetic algorithm which spent 295,000 ms for 256 members and 120 numbers of cities for the Traveling salesman problem run on the SPLASH 2 fabric.

Chapter 3

The initial implementation of the hardware-based genetic algorithm

3.1. The hardware genetic algorithm

Scott et al. suggested a hardware-based genetic algorithm using VHDL. The entire system consists of two parts. The first part called front end is a program in C language running in a UNIX environment on a host computer. The second part, called back-end, contains the implementation of the genetic algorithm [10], [23].

The hardware genetic algorithm consists of seven ports:

- Memory interface module (MIM)
- Population sequencer (PS)
- Random number generator (RNG)
- Selection module (SM)
- Crossover and Mutation module (CMM)

- Fitness module (FM)
- Shared memory (MEM)

After loading the initial parameters into the shared memory, front-end part signals back-end part with a "go" signal. The memory interface detects the "go" signal and initializes the population sequencer, the random number generator, the crossover and mutation and the fitness module. Each of these modules requests appropriate user – defined parameters from the shared memory, and the memory interface module fetches them and transmits them. The population sequencer starts the pipeline by requesting population members from the memory interface module and passes them to the selection module. The selection module receives the members and decides whether a member will be selected according to a Roulette wheel selection algorithm. If this member is selected, the selection module waits for another member to be selected and pairs these members. After mating, selection module sends the pair to crossover and mutation module and resets itself. The crossover and mutation module receives the two members and decides whether it will apply crossover and mutation based on a random number sent from the random number generator. Finally, the new members are sent to fitness module where they are evaluated by the appropriate fitness function. After the completion of this process, fitness module writes the two members into the shared memory via the memory interface module. The steps described above are executed until the hardware genetic algorithm is finished. This decision is taken by the fitness module which determines the current state of genetic algorithm. As soon as the algorithm finishes, the fitness module notifies the memory interface module which sends the "done" signal to the front-end part. Then the user reads the final population from the shared memory.

3.2. The platform

The genetic algorithm was implemented on a BORG board connected on a PCI bus of a host Computer. BORG board contains five FPGA's:

- Two xc4003S containing user-specified logic
- Two xc4002S containing user-specified interconnects between the xc4003s
- One xc4003 for controlling the interface to the PCI bus

Moreover it contains 8 Kbytes SRAM and an 8 MHz oscillator. Because the board did not fit the entire algorithm, additional FPGA's were inserted on the BORG prototyping area. The pseudo random number generator and the crossover and mutation module were placed on an XC4003S FPGA. Furthermore, the memory interface and the population sequencer were shared one of the three xc4005S FPGAs of the prototyping area. The other two XC4005S house the selection module and the fitness module. In Figure 22, 23 two schematics of the BORG board and its prototyping area are presented.

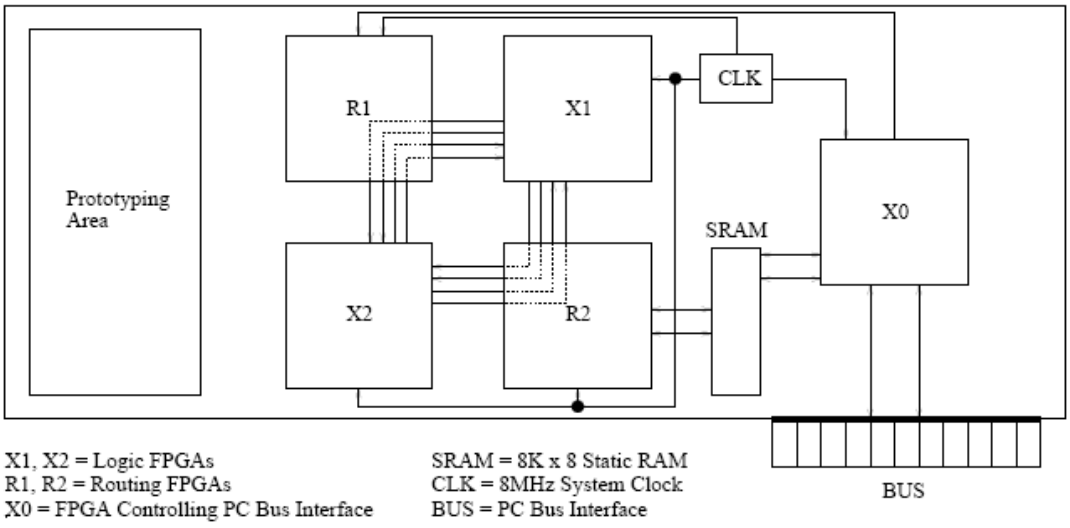

Figure 22: Schematic of the Xilinx BORG prototyping board

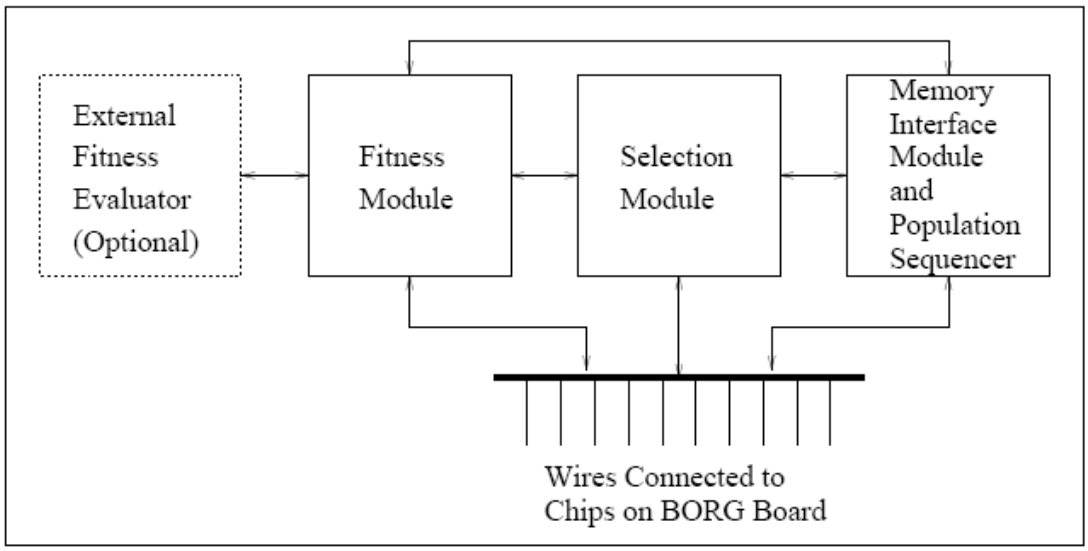

Figure 23: Schematic of the FPGA in the BORG board's prototyping area

3.3. The modules and their operations

In this section the operations of the modules of the hardware genetic algorithm are presented in detail. In Figure 24, we present the block diagram of the initial implementation of the genetic algorithm. The modules communicate via a simple asynchronous handshaking protocol. When transferring data from the initiating module *I* to the participating module *P*, *I* signals *P* by raising a request signal to "1" and waits an acknowledgement. When *P* agrees to participate in the transfer, it raises an acknowledgement signal. When *I* receive the acknowledgement, it sends to *P* the data to be transferred and lowers its request, signalling *P* that the information was sent. When *P* receives the data, it no longer needs to interact with *I,* so *P* lowers its acknowledgement. This signals *I* that the information was received. Now the transfer is complete and *I* and *P* are free to continue processing.

3.3.1. The memory interface module

The MIM provides a transparent interface to the shared memory for the rest system. It consists of the control of the overall design because it initializes other modules as soon

as it detects the "go" signal from the front-end. The population sequencer, the random number generator, the crossover and mutation and the fitness module request parameters from memory interface module by raising a request signal and awaiting an acknowledge from it. As soon as they receive the acknowledgement, each of them sends the address of the parameter it needs to the memory interface module which fetches the data from the specified address of the shared memory. The parameter received from memory is then passed to the requesting module.

As mentioned above there are two populations in the shared memory, the current population from which the population sequencer reads and the next population where the fitness module writes. Memory interface module examines a signal "toggle" driven by the fitness module which specifies which population is being read from and which is being written to. This value is toggled by fitness module after every generation [20].

3.3.2. The population sequencer

The PS requests the population size from the memory, by raising a request to the memory interface module. The memory interface module accepts this transaction by sending an acknowledgement to the population sequencer. Furthermore, population sequencer after receiving the acknowledgement it sends the address of the parameter it needs from the shared memory to memory interface module and MIM fetches it from the shared memory. After reception of the parameter i.e. population size, it repeats the process to get the first member from the shared memory. Then it passes it to the selection module and waits for the next member from the shared memory [20].

3.3.3. The random number generator

The RNG generates a sequence of pseudorandom bit strings based on the theory of cellular automata. Cellular automata generate better random bit strings than linear feedback shift register (LFSR) [28], [29]. The cellular automata consist of 16 alternating

cells which change their states based to rules 90 and 150 [20]. The meaning of these rules is given below:

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

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

In the above equations, S_i is the current state of cell i in the linear array and S_i^+ is the next state for S_i. Rule 90 changes S_i state according to its neighbours only, but Rule 150 takes care of its own state as well when updating. In this implementation the sequence 150 – 150 – 90 – 150 ...90 – 150 that produces more randomness than LFSR because it cycles through all the possible bit patterns 2^{16} except all 0's. Moreover the random number generator is a key component of the HGA system as its outputs are used by SM and the CMM. RNG feeds the SM with random numbers for scaling down the sum of fitnesses of the current population. This scaled sum is used during the selection of members from the population. In addition it supplies CMM with random numbers for determining whether to perform crossover and mutation, and for choosing the crossover and mutation points.

3.3.4. The selection module

The SM utilizes the roulette wheel technique found on the software-based genetic algorithm. The difference is that Scott implementation selects a pair of members A and B, simultaneously instead of one member selected by software genetic algorithm. Initially, it receives the sum of fitnesses of the current population from FM and then two random real numbers are received from the RNG in order to scale down this sum of fitnesses [20].

$$S_{scale} = r_a \times S_{sumoffitnesses}$$

$$S_{scale} = r_b \times S_{sumoffitnesses}$$

Furthermore each member of the population is examined for selection and its fitness value is stored in a running sum of fitnesses Sra. If $S_{ra} > S_{scale}$ then the member under examination is selected. The same process is performed for member B. When the selection module has the random pair of members it signals the CMM module with a request and if receives an acknowledgement it forwards the selected members to the CMM module. After this process finishes, it resets itself and waits for more input. In each generation the SM receives the new sum of fitnesses and repeats the above process until the end of the algorithm.

3.3.5. The crossover and mutation module

When the algorithm starts, the CMM module requests from the MIM the crossover and mutation probabilities with the handshaking protocol described above (§ 3.3). These probabilities are specified by the user and placed in the 2nd and 3rd position of the shared memory. Moreover the CMM receives the selected members from the SM, and four random numbers from RNG. It compares the first random number "$rand_1$" with the crossover probability "P_c" and if "rand1<P_c" then crossover is performed between the selected members resulting into two new members A' and B'. A second random number "$rand_2$" is received indicating the crossover point. If $rand_1 \geq P_c$ then the two members A and B are copied to the new members A' and B'. After the crossover operation is finished, the mutation operation is performed. The random number $rand_3$ is compared with the mutation probability and if $rand_3 < P_m$ then mutation is performed on a single bit of A' indicated by the random number $rand_4$.The new members A' and B' are then forwarded to the FM for evaluation by the fitness function [20].

3.3.6. The fitness module

The scope of the FM is to evaluate the two members received from the CMM, with defined fitness function, and write them to the appropriate locations in the shared

memory via the MIM. As soon as the algorithm starts, it sends a request to the MIM to fetch three genetic parameters from the memory: 1) population size, 2) sum of fitnesses and 3) number of generations. Then it receives a request from the CMM module to get the two members for evaluation. The FM incorporates the fitness function F(x) = 2x only, which evaluates each member's fitness value in a single clock cycle. Moreover, the FM has the knowledge of the current sum of fitnesses of the population, and sends the value to the SM after each generation. If the algorithm runs for the specified number of generations, then FM informs the MIM and sends a "done" signal to the front-end to inform it that algorithm is finished [20], [23].

3.3.7. The shared memory

Shared memory is the medium where the genetic parameters and the initial population are stored in. It is not an actual part of the hardware genetic algorithm but is presented here for further understanding. Before the algorithm starts, the front-end writes the parameters and the initial population to this memory which is shared with the back-end. The algorithm starts and the MIM sends the appropriate parameters to the appropriate modules. The PS reads the current population and the FM writes the next population during a run. Finally, when the genetic algorithm is finished, the final population is stored in the shared memory. [23].

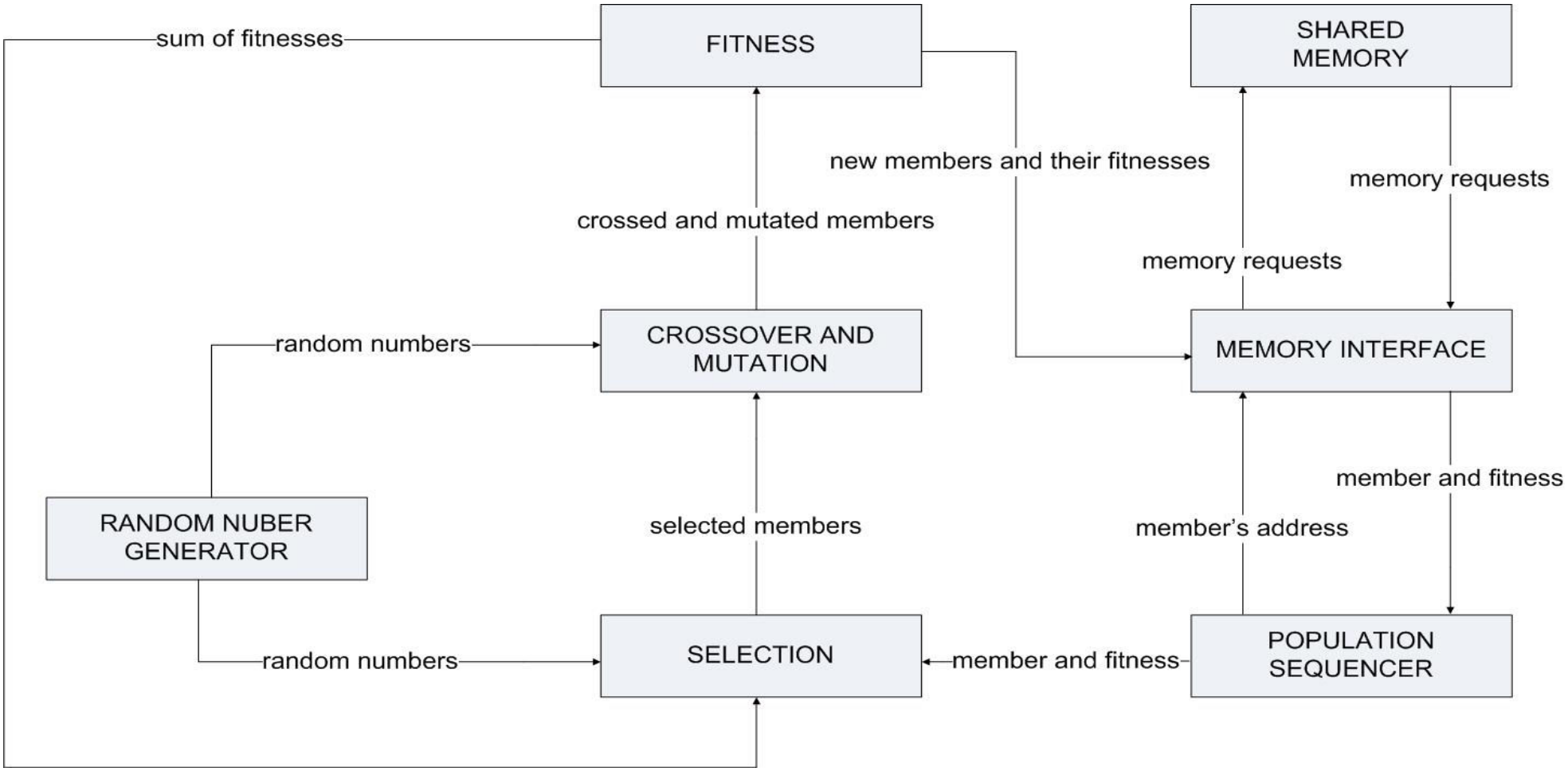

Figure 24: Block diagram of the genetic algorithm

3.4. Pipeline and parallelization

The modules defined above operate concurrently with each other and form a coarse-grained pipeline with 5 stages (Figure 25). The PS starts the pipeline by requesting members from the MIM, passing them to the SM. Selection module decides if it selects a pair of member according to roulette wheel method and then passes the pair to crossover and mutation module and restarts itself, awaiting new members from population sequencer. Crossover and mutation module after processesing the members it received, passes them to fitness module for evaluation and waits for a new selected pair of members. Finally, fitness module evaluates the two members and writes them to the shared memory [23].

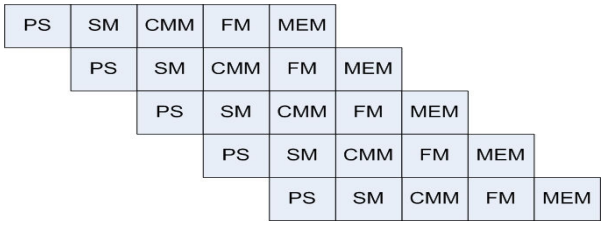

Figure 25: The stages of the pipeline

If sufficient chip area is available then different types of parallelization can be added (Figure 26). Multiple selection modules can be inserted for speeding up the selection process. The original code provides the option to use a second selection module. However this inserts complexity to the design because the PS should pass members to both the SMs and then the CMM must check the two SM to receive requests. Moreover to extend the parallelism, selection, crossover and mutation and fitness pipeline can be replicated several times in order to form parallel pipelines. Finally the highest degree of parallelism can be achieved by inserting several numbers of selection, crossover and mutation and fitness modules. Each selection would be connected to each crossover and mutation and each crossover and mutation to each fitness module.

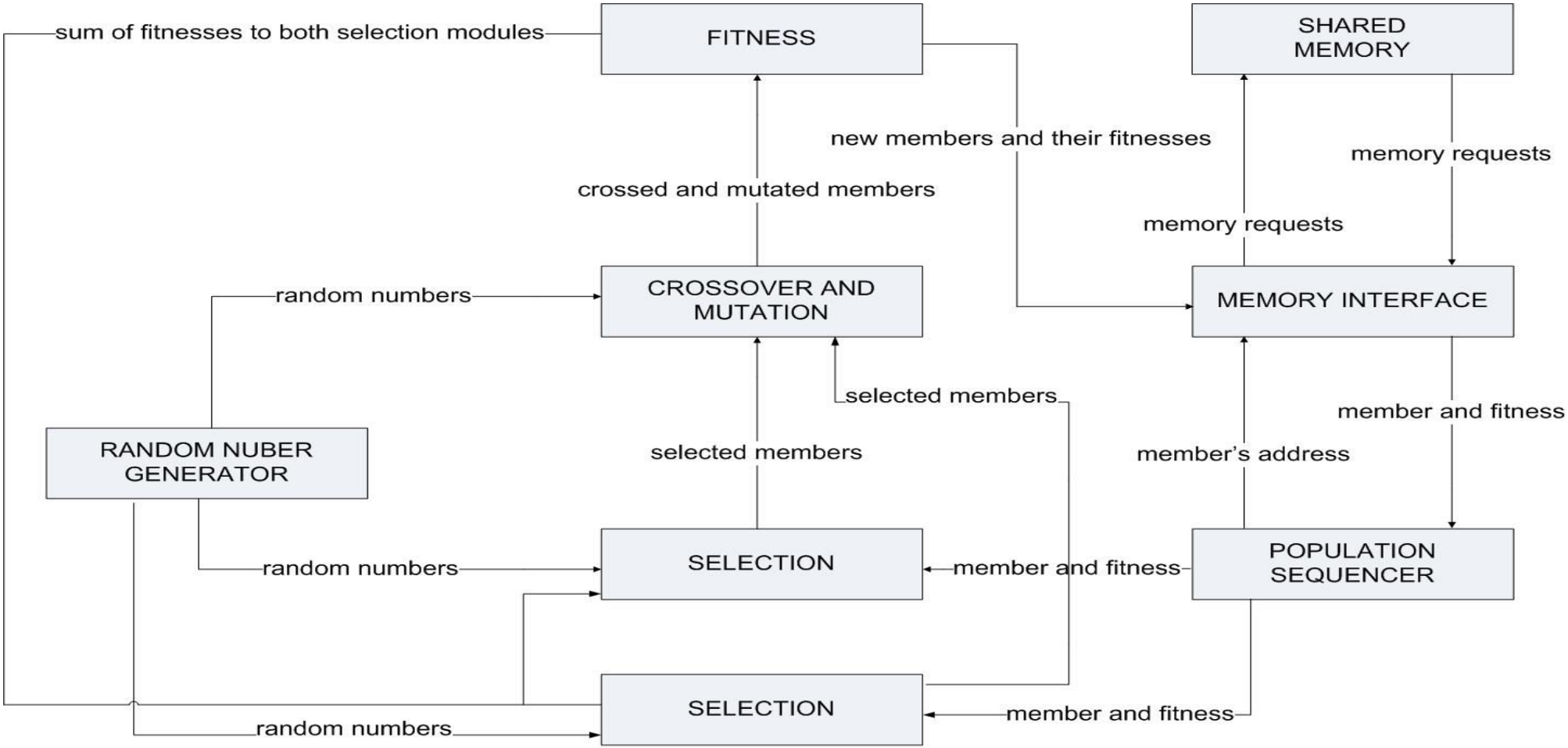

Figure 26: Parallel selection modules

3.5. Scalability of the design

All the modules above are written in VHDL language within the Mentor Graphics Environment [10]. There is also a package file where the number of bits of some basic parameters is placed. This allows for easy modifications on the size of some basic parameters which would then be applied on the entire design. Some basic parameters are: i) the maximum width in bits of the population members (n), ii) the maximum width in bits of the fitness value (f), iii) the maximum size of the population (m) and iv) the maximum number of generations (g).

3.6. The C code of the host computer

As mentioned above the application with C that runs on the host computer is responsible for producing the input file to the system. User determines the fitness function which will be applied on each member of the population. In addition, some parameters are also defined by the user such as the member width (n), the fitness value width (f), the number of generations (g), the crossover and mutation probabilities, the population size (m), the memory word's width and memory size. The output of this program is a text file with 128 lines:

At the first six lines (0 - 5) of the file, 6 genetic parameters are placed:

- Population size
- Random number generator seed
- Mutation probability
- Crossover probability
- Sum of fitnesses of the first population
- Number of generations

In the next 32 lines (6 - 37) the initial population is placed the first population. Then lines (38 - 69) represent the generated population and finally lines (70 – 127) are unused.

All parameter values are represented in binary and interpreted as unsigned integers. Moreover, each entry in the population consists of n bits for the member followed by f bits for its fitness. On the other hand, the back-end is the hardware where the genetic algorithm runs on. These parameters and the initial population are loaded into a memory which is shared between the front-end and the back-end.

The program generates the initial population according to the fitness function to be used. More specifically the latter affects the width of the members. For example, if n=4 and the function to be used is F(x)=2x, then f=5 and thus the width of the members will be n+f=9 bits. The initial population is generated randomly with a "rand" function used into the C code. However, for a specific problem the initial population members can be inserted by the user.

3.7. Results

The hardware-based genetic algorithm proposed by Scott was compared against the software genetic algorithm implemented on a Silicon Graphics 4D/440 with four MIPS R3000 CPUs each running at 33 MHz. The first 6 tests were run on the prototype while the last 6 tests were run on a VHDL simulator. All I/O times were removed from the comparisons. The HGA prototype used an average 6.8 % as many clock cycles as the SGA and the execution time grows quadratically with m and linearly with g. Furthermore, the tests on the prototype utilized a population size m = 16, member width n = 3, two SMs and fitness value width f = 4. The tests on the VHDL simulator used m = 32, n = 4, one SM and f = 12. The results are presented in the Table 20. The first six tests refer to the BORG board whereas the rest refer to the simulation.

Fitness Function F(x)	Number of Generations	SGA Clock Cycles	HGA Clock Cycles	HGA Speedup(Cycles)
x	10	97,064	5,636	17.222
x	20	168,034	10,622	15.819
x + 5	10	99,825	5,585	17.874
x + 5	20	170,279	10,945	15.558
2x	10	101,019	5,390	18.742
2x	20	170,241	10,659	15.972
x^2	10	334,210	22,892	14.599
x^2	20	574,046	45,019	12.751
$2x^3 – 45x^2 + 300x$	10	342,806	22,892	15.178
$2x^3 – 45x^2 + 300x$	20	589,863	44,503	13.254
$x^3 – 15x^2 + 500$	10	333,701	21,362	15.621
$x^3 – 15x^2 + 500$	20	579,176	44,317	13.069

Table 20: Performance of the SGA and the HGA

It is worth noticing that the clock of the prototype was 2 MHz because the glitches or noise in the wire wrapped part of the prototype prevented faster clocking.

Chapter 4

Implementation of a hardware-based genetic algorithm on a Virtex II Pro FPGA

4.1. Resources utilization analysis for the implementation on the XUP platform

Before proceeding with the implementation we checked whether the initial implementation could be accommodated by the XUP platform. Moreover, we made a comparison between the available resources of the BORG and the XUP boards. Regarding the BORG board no published information exists regarding the resources utilization of the HGA. Therefore, we only present information on the available resources of the FPGAs mounted on the BORG board that were used for the HGA. As described in the Chapter 3, BORG board and its prototyping area contains eight Xilinx FPGAs. Originally the BORG board contains two XC4003S, two Xilinx XC4002S and one Xilinx XC4003, whereas on the BORG prototyping area three more XC4005s

FPGAs were placed. More information on the resources of the above FPGAs is given in Table 21. The total number of CLBs is 1016.

Device	Logic Cells	CLB Matrix	Total CLBs	Number of Flips Flops	Max Logic Gates	Max RAM bits	IOBs	Program data
XC4002s	152	8 X 8	64	256	1,600	2,048	64	61,052
XC4003/XC4003s	238	10 X 10	100	360	3,000	3,200	80	53,936
XC4005s	466	14 X 14	196	616	5,000	6,272	112	94,960

Table 21: Resources analysis of BORG board and its prototyping area

Tables 22 and 23 have the resources analysis of the XC2VP30 FPGA of the XUP [21]. It contains 3424 CLBs which is much bigger than the number of CLBs in BORG board. Moreover, the CLB unit of the XC2VP30 is more complex and carries more logic as compared to the above devices. Therefore, the initial implementation of the genetic algorithm can be easily fit into the XC2VP30 FPGA.

Device	Rocket IO Transceiver Blocks	PowerPC Processor blocks	Logic Cells	Slices	Max Distributed RAM	18 X 18 Multiplier blocks	Maximum user I/O Pads	DCMs
XC2VP30	8	2	30,816	13,696	428 Kbits	136	644	8

Table 22: Resources analysis of the Virtex II Pro FPGA (1)

Max Block RAM	Configuration Bits	CLB Array: Row X Column	Number of Carry Chains	Number of SOP Chains	BRAM 18Kb Blocks	Flip Flops	LUTs	Number of 3-State Buffers
2,448Kbits	11,589,920	80 X 46	92	160	136	27,392	27,392	6,848

Table 23: Resources analysis of the Virtex II Pro FPGA (2)

4.2. The Virtex II Pro FPGA

The FPGA used was the XC2VP30 FPGA of the Xilinx Virtex II Pro family. It is organized as a column based array of logic elements. The basic element is the configurable logic block (CLB) which contains look-up tables (LUT) as the basic function generators. The XC2VP30 FPGA device has an 80x46 CLB array. Each CLB has four slices and two three-state buffers. Each slice has two function generators, f & g, two storage elements and arithmetic logic gates. The architecture can be characterized as fine-grained due to the small function generators which can be programmed in bit-level. Moreover this FPGA family is hybrid due to the several specialized circuits such as Block SelectRAM (BRAM) resources, multiplier blocks and Digital Clock Manager (DCM) modules [22]. The perimeter of the FPGA is occupied by Input/Output blocks (IOB) which are responsible for managing the FPGA pins. All logic is connected via programmable routing resources organized in a hierarchical Global Routing Matrix (GRM). In addition, two hard core IBM PowerPC 405 processors are incorporated into the FPGA fabric.

4.3. Modifications for porting the HGA on the Virtex II Pro FPGA

The initial implementation of the HGA was implemented in VHDL within the Mentor Graphics environment [26]. We proceeded with modifications on the code and its porting using Xilinx ISE 7.1 environment. In order to perform this, we initially removed the *mgc_portable* library which was supported in the Mentor Graphics environment only, and we then replaced the *qsim_state* and *qsim_state_vector* data types with the identical *bit* and *bit_vector* data types. This was also suggested in Scott's work [20]. Moreover there were two types of functions in the VHDL code such as *to_qsim_state(i,s)* and *to_integer(r)*. The first function, i.e. *to_qsim_state(i,s),* converts an integer *i* to *qsim_state_vector* of s bits wide. The second function, i.e. *to_integer(r)*, converts a *qsim_state_vector* to an integer. These two functions have been replaced with the *to_bitvector* and *to_integer* functions respectively, which are supported by Xilinx.

Furthermore, the implemented shared memory is intended for simulation purposes only, as it supports read and write functions from/to a text file and is not synthesizable. Therefore, the non-synthesizable shared memory was removed and we generated a structural dual port memory, using Xilinx Core Generator, with 128 entries of 9 bits each. The 4 bits concern the member width, and the following 5 bits concern the fitness value. It is worth noticing that the dual port memory generated with the Core Generator supports *std_logic* and *std_logic_vector* data types, so the *bit* and *bit_vector* data types were substituted with the *std_logic* and std*_logic_vector* data types respectively. The shared memory now stands between the PowerPC and the HGA core, and both can access it for reading/writing purposes, each from a different port.

Firstly, we compiled and simulated the VHDL code using Modelsim 6 SE and we verified that the design worked properly. Then we synthesized, and implemented the HGA core with the Xilinx ISE 7.1. Finally, we incorporated the PowerPC processor and we downloaded the design on the XC2VP30 FPGA using the Xilinx EDK 7.1. Table 24 has the resources utilization after place and route.

Number of Slice Flip Flops	1,557 out of 27,392	5%
Total Number 4 input LUTs	2,380 out of 27,392	8%
Number of PowerPC405s	1 out of 2	50%
Number of Block RAMs	9 out of 136	6%
Number of MULT18X18s	1 out of 136	1%
Number of GCLKs	2 out of 16	12%
Number of DCMs	1 out of 8	12%
Number of External IOBs	4 out of 556	12%
Number of LOCed IOBs	4 out of 4	100%
Number of BUFGMUXs	2 out of 16	12%
Number of 4 input LUTs (Logic)	2,061 out of 27,392	7%
Number of SLICEs	1,689 out of 13,696	12%

Table 24: Resources utilization of the XC2VP30 FPGA for the HGA with F(x) = 2x

The BRAM resources are allocated by the main memory of the PowerPC we selected during setup of the processor, and the shared memory standing between the PowerPC and the HGA core. The maximum clock frequency on which the design operates is 100 MHz. It should be noticed that the original implementation contains only the fitness function F(x)=2x in behavioural code. Thus the resources utilization is kept low in this implementation. In the next phase of the work we incorporated more fitness functions.

4.3.1. Fitness functions and their implementation with DSPs

Due to the limited FPGA resources, Scott implemented the fitness function F(x)=2x only on the BORG board. However, he simulated the genetic algorithm with more fitness functions which are shown in Table 25.

Number	Fitness Function F(x)
1	$F(x) = x$
2	$F(x) = x + 5$
3	$F(x) = 2x$
4	$F(x) = x^2$
5	$F(x) = 2x^3 – 45x^2 + 300x$
6	$F(x) = x^3 – 15x^2 + 500$

Table 25: Fitness functions

In order to support the above fitness functions we added a new module which we call fitness evaluation module. It consists of five multipliers and four adders, produced with the Core Generator, a multiplexer and a functional controller. The multipliers have been implemented in a pipelined manner - the *maximum pipeline* parameter was enabled in the Core Generator - with output latency 1. The adders are non-pipelined. The multiplexer selects the appropriate output according to the fitness function that is used. Finally, the functional controller controls the *RDY* signals of the multipliers to inform the FM when the fitness value of the evaluated member is ready.

Several modifications have been made to incorporate all the fitness functions. In order to support larger fitness values the width of bits of the fitness value was changed from 5 to 14. This is illustrated in Table 26. Furthermore, due to the change of the fitness value width, modifications have been made to parameters such as the memory data width and the sum of fitnesses width. Some changes have been also made on the FM of the initial design. More specifically, in the initial implementation the fitness function calculated the fitness value of each member in a single clock cycle. In the new implementation, due to the pipelinining of the multipliers, the fitness value of each member needs more than one clock cycle to be calculated. Moreover, each multiplier has a dedicated output, the *RDY* signal, which indicates when the result is ready. Therefore, the evaluation of the fitness values of members is performed in more than one clock cycle depending on the fitness function and it finishes after the activation of *RDY* signal. The fitness evaluation module is shown in Figure 27 and is described later in more detail.

	Member n	Fitness Function F(x)	Fitness Value f	Member & Fitness Value (n &f)
Scott	1101	2x	11010	110111010
Our Approach	1101	$2x^3 – 45x^2 + 300x$	00001010110001	110100001010110001

Table 26: Example of parameters width

The above design can be slightly changed to support even more fitness functions. A generalized fitness function of the form ax^3+bx^2+cx+d can be used for evaluation, in which the coefficients would be valued externally by the user. Although this entails minor effort it has not been implemented and it is planned as future work. We should also notice that there is a limitation on the maximum size of the polynomial coefficients that present implementation allows, as shown in Table 27.

Polynomial coefficients	Maximum coefficients width	Maximum decimal value
a	6	63

b	6	63
c	12	4095
d	14	16383

Table 27: Maximum sizes of the polynomial coefficients

As the operation is applied on a pair of members, two components of the fitness evaluation module have been instantiated. Therefore, when the CMM module outputs the two new members, i.e. the offsprings, it sends them to the FM and the two fitness evaluation modules. More specifically, both are sent to the FM, and in parallel, each of them is separately sent to a fitness evaluation module. This is performed in order to start the evaluation of the fitness values of the two members in parallel as soon as they are ready from the CMM module. In the initial implementation this process is performed differently. More specifically, when the new members are ready from the CMM module they are passed to the FM where the fitness values are evaluated in a single clock cycle without an external fitness evaluation module. In our implementation we decided to insert these two fitness evaluation components in order to support more complex fitness functions.

At this point we explain the functionality of the fitness evaluation module presented in Figure 27. From the Table 25, assume that the fitness function to be optimized is the first one. Then the input of this module will drive its output directly, i.e. input equals output. Regarding the second fitness function, the output is taken by the F2 given by *ADDER 4* of the Figure 27. All the outputs of the above fitness functions are connected to a multiplexer, and in this way the user selects the fitness function to be optimized. In the present version this is performed by changing a value in the PowerPC code. Thus the selector signal of the multiplexer in the HGA core is driven by the PowerPC. In the next version of the system it is suggested this value to be given externally from an application running on the host computer.

We should mention that after the incorporation of the fitness evaluation module in the HGA core, due to misleading observation of the clock frequency that the synthesis

report, we decided to utilize a Digital Clock Manager (DCM) in order to adjust it (§ 4.4.2). Although the synthesis report produces the clock of the design it gives inaccurate results. After the place and route timing report of Xilinx EDK 7.1 tool we realized that the clock was 100 MHz. As a consequence, we threw out the DCM.

4.3.2. Multipliers implementation

As it mentioned above the multipliers were generated from the Core Generator. We tested all the possible configurations with Core Generator, in order to realize which gives us the maximum frequency. Moreover the Core Generator of Xilinx ISE 7.1 didn't give us the opportunity to select the output latency of the multipliers, which can be done with the Core Generator of Xilinx ISE 9.1 design tool. The results are presented in Table 28.

i	Maximum pipeline	Minimum pipeline	Output latency	Register input	Register output	Maximum Frequency of overall design (MHz)
1	√	—	1	—	—	101
2	√	—	2	√	√	102
3	√	—	1	√	—	102
4	√	—	1	—	√	105
5	—	√	0	—	—	60
6	—	√	2	√	√	94
7	—	√	1	√	—	98
8	—	√	1	—	√	99

Table 28: Performance of multipliers

It can be observed that multipliers implemented with maximum pipeline run faster than multipliers implemented with minimum pipeline. The last column indicates the overall

design's maximum frequency. The maximum frequency was achieved utilizing the fourth multiplier of the Table 28 and the overall minimum frequency was achieved using the fifth multiplier. So the five multipliers that are placed in each fitness evaluation module were produced with maximum pipeline and output latency 1.

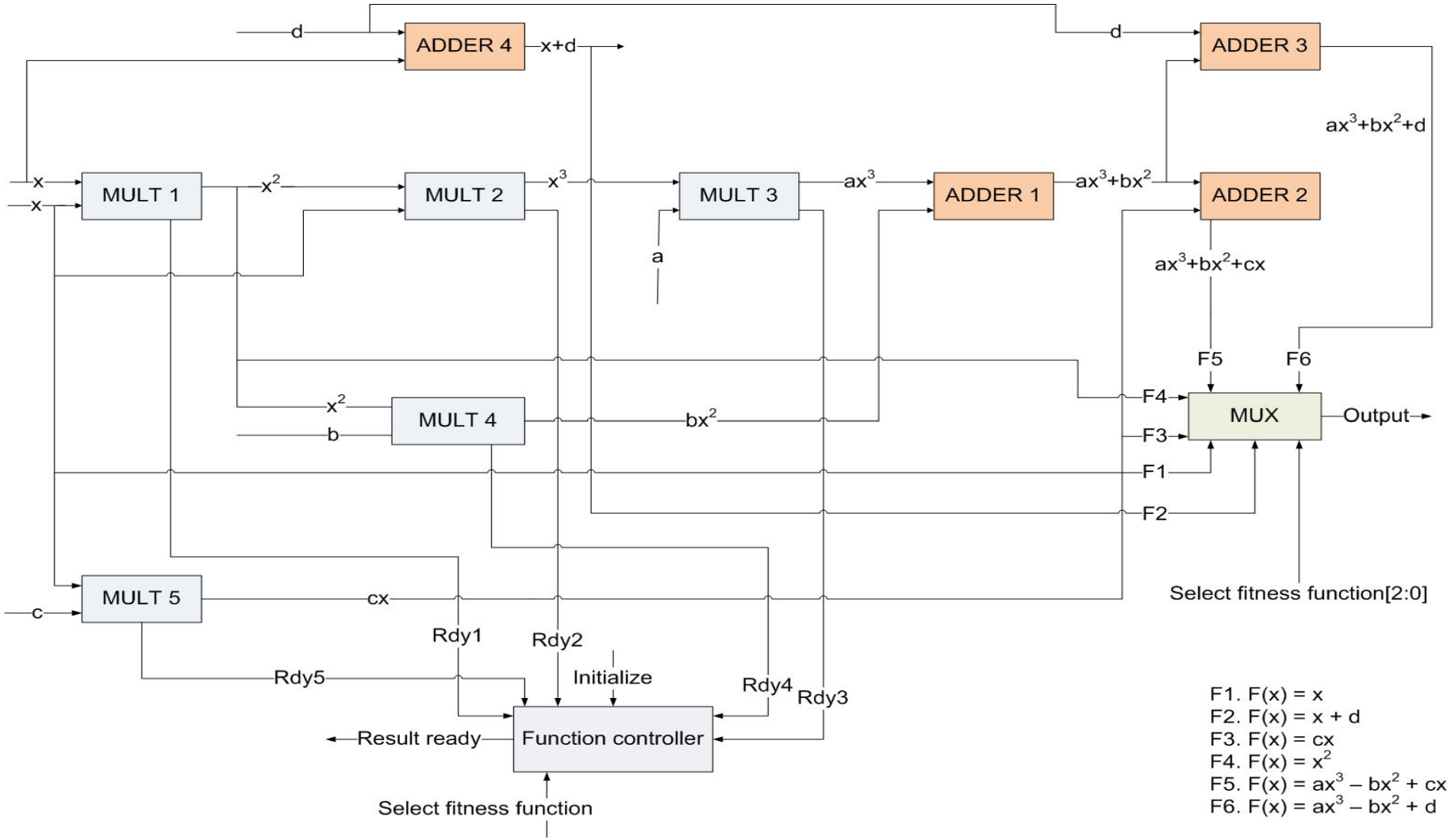

Figure 27: Fitness functions with DSPs

4.4. The PowerPC and the Software

Until now, the system description has been focused on the hardware side of the FPGA system. The master peripheral of the FPGA is the PowerPC, meaning that is the only one that can initiate communicates. The PowerPC receives the data from UART and stores them into a table (array) in its BRAM. Furthermore, we give inputs to our system by writing appropriate values to the memory mapped registers. For example, if we want to start the genetic algorithm in the FPGA we must set the input *go* signal with the value "1". The PowerPC writes to the *register 0,* which is connected with the input *go,* the value 1, using the function XIo_Out16. If we want to read a value from an input or an output we use the *XIo_In16* function. Below the two functions for read and write operation are shown.

XIo_Out16 (XPAR_HGA_0_BASEADDR +0x0, 1)→ this is for the write operation

Short Register0=XIo_In16 (XPAR_HGA_0_BASEADDR + 0x0)→ this is for the read operation

After initializing the inputs of the genetic algorithm, the PowerPC sends the data from the table (array) to the shared memory and activates the *go* signal. When the algorithm is finished, the *done* output becomes high. When this is detected from the PowerPC, the latter reads the data in the shared memory and stores them in a table (array). Then the data are sent to the host computer via hyper terminal, for the user to read the final population.

4.5. The embedded system

An application in C running on UNIX environment on a host computer generates the text file with the genetic parameters and the initial population. The values of this text file must be then inserted in the shared memory. In order to send these values from the host computer to the shared memory implemented in the XC2VP30 FPGA we used

the RS232 port of the XUP. For this reason the windows hyper terminal was used and the PowerPC 405 embedded processor was utilized for UART communication.

The text file binary representation was converted to hexadecimal, and then to ASCII with a program in C [23]. Because the hyper terminal can send 1 byte at a time, and the width in the shared memory is 18 bits, the data were sent in the following way: we send the first 8 bits and then we send the second 8 bits. The remaining 2 bits are sent within a byte and its LSB is filled in with six zeros. An example is shown in the Table 29. If the 110100000000011010 bit string is to be sent to the shared memory, it is segmented in three bytes and their ASCII values, D0, 06 and 80 are sent. Inside the PowerPC there is a code segment that recovers the original bit string, and then sends it to the shared memory.

	Row in text file	First byte	Second byte	Third byte
Binary representation	110100000000011010	11010000	00000110	10000000
Hexadecimal representation	3401A	D0	06	80

Table 29: Representation of input file

The FPGA system is illustrated in the Figure 28 and is formed by a number of components and their interconnections. The components can be distinguished into two categories, buses and peripherals.

Two different buses are available:

- Processor Local Bus (PLB). It is the system's high speed bus and is intended for communication between high speed peripherals.
- On-chip Peripheral Bus. It is the system's low speed bus and is intended for use by peripherals where there is no need for high-speed communications.

- PLB to OPB BRIDGE. This component connects the OPB bus with the PLB bus.

The following peripherals have been instantiated:

- PowerPC405. It is the hard core of an IBM PowerPC processor. It is a PLB master and its maximum frequency of operation is 300 MHz. The program code is executed in this peripheral.
- PLB_BRAM block. This is the system main memory. All the program sections are located in this memory.
- PLB_BRAM_if_cntrl. This peripheral is a PLB slave and controls the BRAM.
- DCM module. This component synthesizes the processor's 300 MHz frequency from the board's 100 MHz clock.
- Proc_sys_reset. This component creates the necessary reset signals.
- opb_uarlite. This is an OPB slave peripheral for handling the serial communication with the personal computer.
- HGA core. This is a custom OPB slave peripheral where the genetic algorithm is placed on. The top module of our design is instantiated in the user_logic module where the inputs and outputs signals are connected to memory mapped registers. Moreover, the user_logic is instantiated in the HGA core which is connected to the opb_bus.

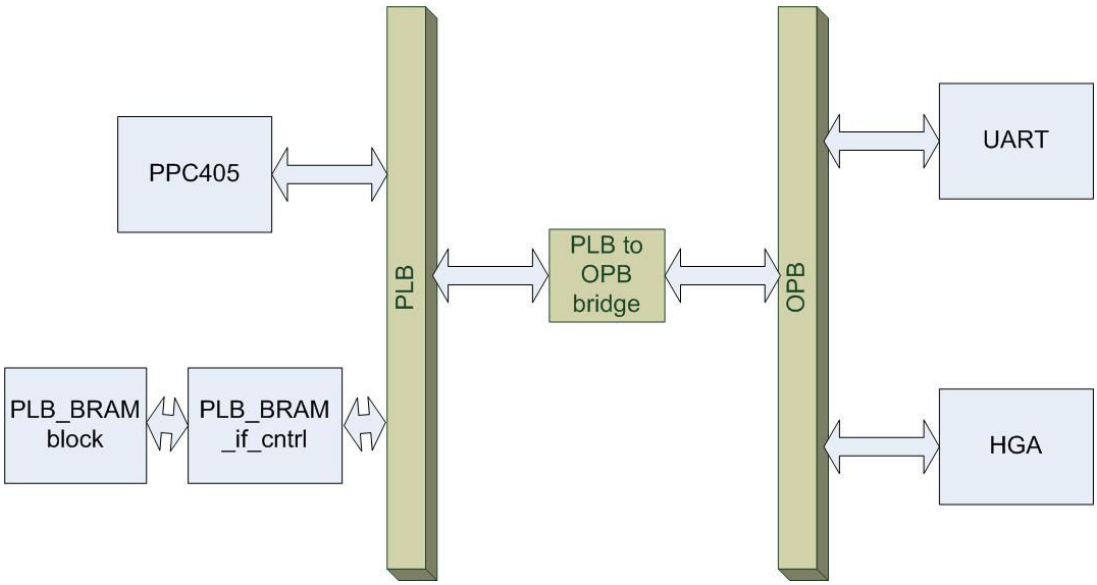

Figure 28: The FPGA System

4.5.1. System Operation

The flowchart of the system operation is presented in Figure 29. Firstly, the user selects the genetic parameters and the fitness function to be optimized by the genetic algorithm. Then, the input file, after its conversion from binary to hexadecimal, is transferred via the RS232 of the host computer to the PowerPC. The PowerPC receives the input file and stores it into a table in its BRAM. These data will then be inserted in the shared memory. Then, the PowerPC activates the *go* signal in order to trigger the genetic algorithm and enters an awaiting state until the *done* signal is activated. This indicates the completion of the algorithm execution. Then, the PowerPC reads memory contents and stores them in a table. Finally, the PowerPC transfers the data to the host via the RS232 port and enters an awaiting state. In this state if the user wants to send a new input file the above process is repeated, otherwise it finishes.

We should mention that the Virtex II Pro FPGA is in general a loosely coupled architecture. This is due to the buses and the registers that intervene for the communication between the processor and the peripherals. This results to latencies that can not be predicted by the designer. For this reason, a controller has been

implemented as part of the HGA core which controls the communication between the PowerPC and the peripheral. The controller checks when the address is changed, and when this happens it sets high the enable signal of the memory port A for one cycle. To achieve this, for each memory we want the PowerPC to have access, we inserted one register. This keeps the previous address, in order to compare it with the current one. The two addresses are compared to examine whether a new address appears on the input of the memory address lines.

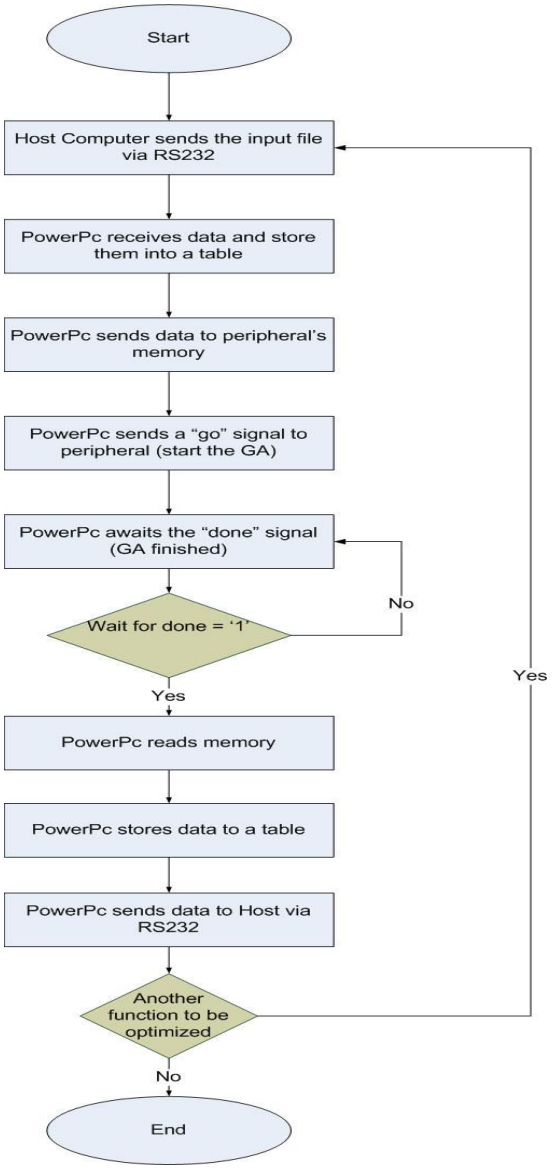

Figure 29: Flowchart of the system operation

4.5.2. Digital Clock Manager (DCM)

The DCM is a component produced with the Core Generator and is used for frequency synthesizing. In our design the DCM takes a clock input of a specified frequency and produces two clock outputs. It takes the input of the OPB clock which is 100 MHz, and it outputs a clock with 100 MHz frequency and a clock with 90 MHz frequency. The 100 MHz clock is connected to the port A and the 90 MHz clock is connected to the port B of the dual port shared memory. The 90 MHz clock also drives the HGA core to work properly. The read and write operations between the PowerPC and the shared memory are performed with a clock of 100 MHz. The read and write operations between the HGA core and the shared memory are performed with the 90 MHz clock. The design with the DCM was downloaded on Virtex II Pro FPGA and its correct functionality was verified. Figure 30 illustrates the DCM.

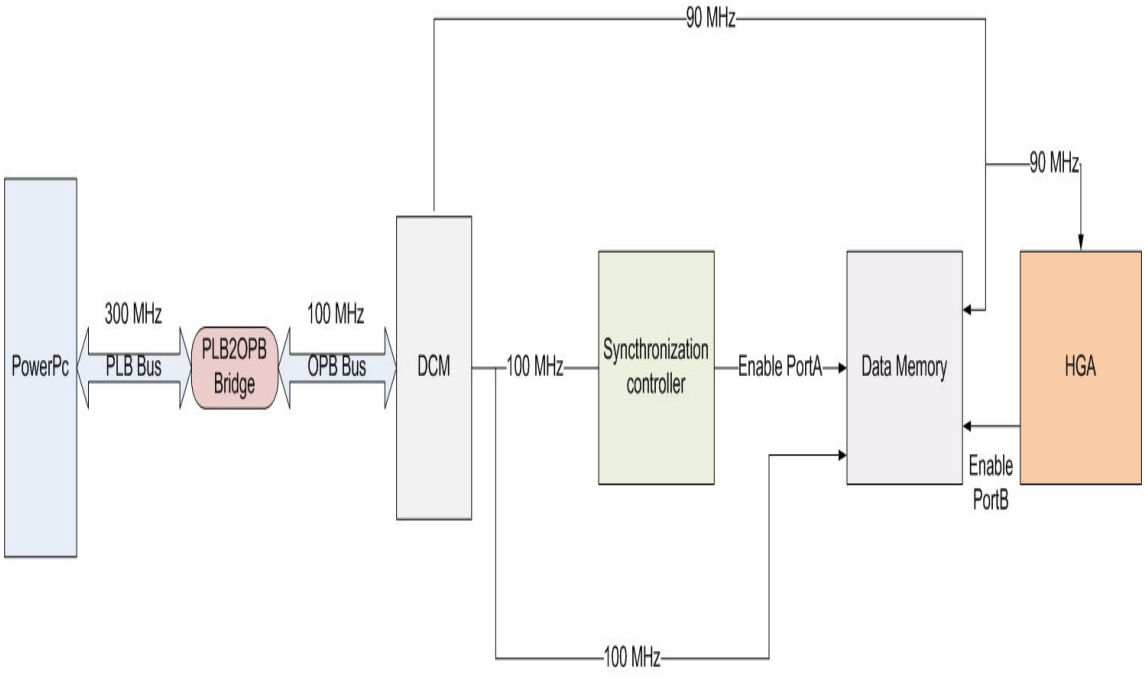

Figure 30: Digital clock manager

4.5.3. Extensions for measurements

After we have simulated and synthesized the VHDL code, extensions for measurements have been made. A dual port memory with 128 entries × 19 bits was generated to store the intermediate sum of fitnesses of each population. By observing the sum of fitnesses we have a quality criterion for the populations that have been generated by the algorithm. This helped us to make our experiments that are discussed in next chapter.

Moreover a new component was included in the overall design to count the clock cycles for each run. It is a counter that counts the clock cycles while *done* signal is set to zero, which means that the HGA core executes. I/O timings have been removed because we are mainly interested in the execution time of the HGA core. Figure 31 represents the block diagram of the HGA after our modifications.

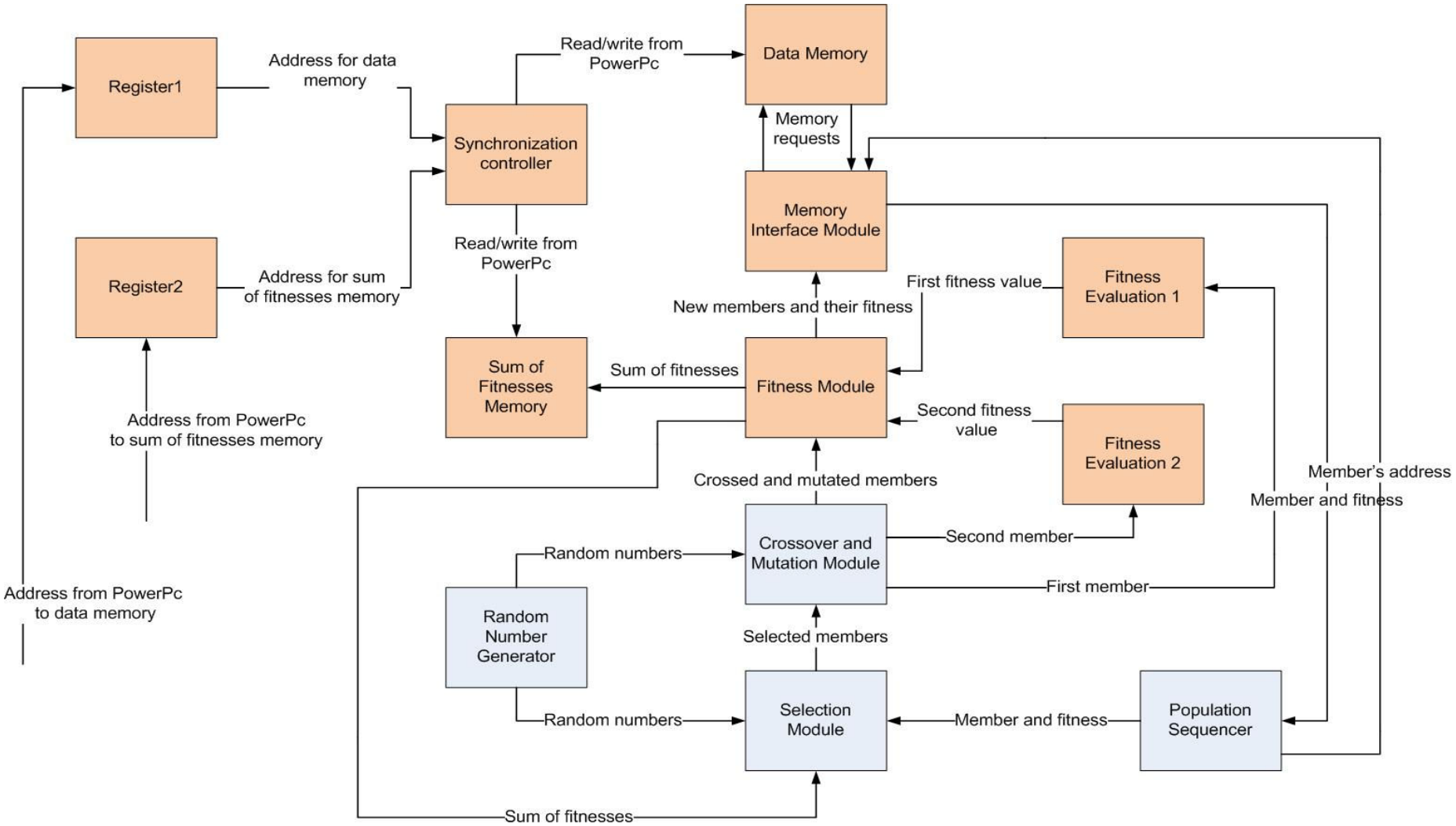

Figure 31: Block diagram of our approach for the HGA

4.6. Implementation results

We have inserted two dual port RAM's; one for the genetic parameters, the initial and the final populations, and the second for the sum of fitnesses of each generated population. The system's clock frequency is 100 MHz. We downloaded the final implementation on Virtex II Pro to verify its correct operation in hardware. The results are shown in Table 30. The minimum period after post place and route synthesis with EDK 7.1 was 100 MHz.

Number of Slice Flip Flops	2,024 out of 27,392	7%
Total Number 4 input LUTs	3,190 out of 27,392	11%
Number of PowerPC405s	1 out of 2	50%
Number of Block RAMs	10 out of 136	7%
Number of MULT18X18s	11 out of 136	8%
Number of GCLKs	2 out of 16	12%
Number of DCMs	1 out of 8	12%
Number of External IOBs	4 out of 556	12%
Number of LOCed IOBs	4 out of 4	100%
Number of BUFGMUXs	2 out of 16	12%
Number of 4 input LUTs (Logic)	2,783 out of 27,392	10%
Number of SLICEs	2261 out of 13696	16%

Table 30: Resources Utilization of HGA with more fitness functions

4.7. Weaknesses of our implementation

After the analysis of our implementation we present here some weaknesses of our design:

- In the VHDL code we download from the internet it was included a package file that supports parameterized and scalable design. In order to download the genetic algorithm in the Virtex II Pro FPGA, we took out this file because

Xilinx ISE 7.1 tool couldn't recognize it. In the other hand Xilinx ISE 9.1 supports package files but we didn't employ it for stability reasons which are provided by the Xilinx ISE 7.1 design tool. In the future, the package can be included to support parameterized and scalable design.

- Another design weakness was that we want to implement the six fitness functions which were presented in Scott Master thesis, so the vector sizing of the coefficients was chosen for supporting these fitness functions. If user wants to implement different fitness functions, he should change the width of the inputs and outputs of the multipliers and the adders. In addition, changes to the VHDL code should be done to the coefficients of the fitness functions which by the way support the 6 fitness functions of Scott Master thesis.

- In current version the selector signal of the multiplexer that controls the output of the fitness evaluation module is driven by the PowerPC. In the next version of the system it is suggested this value to be given externally from an application running on the host computer.

- A generalized fitness function of the form ax^3+bx^2+cx+d is not supported yet. In a future version the coefficients could be valued externally by the user.

Chapter 5

Experimental results and validation

5.1. Experimental results

In order to make the experiments we inserted a new dual port memory to keep the intermediate sum of fitnesses of each generation. Moreover, a counter was implemented to count the clock cycles from hardware implementation of the genetic algorithm.

Six populations were produced from the C code of the host computer for each of the 6 fitness functions described in chapter 4. The experiments were done for population size m = 32, member width n = 4 and fitness value width f = 14. The algorithm runs for the 36 initial populations, and we take results that are presented below in Figure 32.

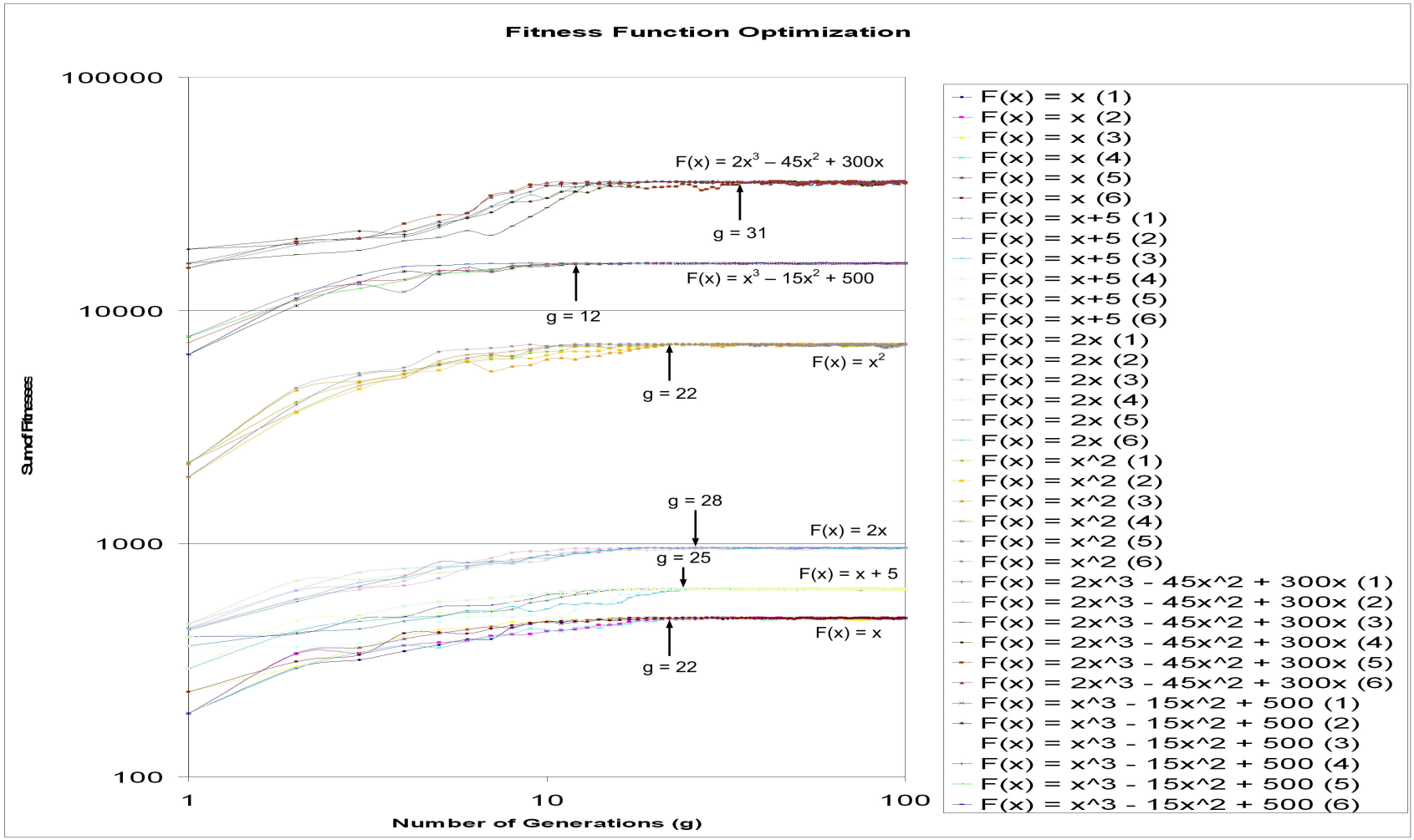

Figure 32: Fitness function optimizations for several populations

As we can see from Figure 32 above the populations are well optimized in a few generations. Fitness function $F(x) = x^3 – 15x^2 + 500$ was optimized in 12 generations while other fitness functions need 20 and 30 generations to be optimized. This probably happens because $F(x) = x^3 – 15x^2 + 500$ is converging to 0 and 15, which are the two optima. Moreover, this convergence was depended on the random number generator, the probabilities of crossover and mutation and most importantly the initial population. We observe that 35 numbers of generations are enough for all the fitness functions to be optimized. Moreover, fitness function $F(x) = 2x^3 – 45x^2 + 300x$, was optimized after 32 generations because it is the most complex function. Table 31 below presents the optimization performance average results, and timing results of the HGA runs on the 6 fitness functions. We observe that the evaluation time differs for each fitness function. This is due to the variations of the initial populations that are generated from the C code running on the host computer for each fitness function.

F(x)	Number of Generations (g)	Sum of Fitnesses (Initial Population)	Sum of Fitnesses (Final Population)	Increase Percentage (%)	Clock Cycles
x	100	201	478	137	218,839
x+5	100	351	639	82	217,947
2x	100	437	937	118	218,225
x^2	100	2121	7180	238	217,363
$2x^3 – 45 x^2 + 300x$	100	16536	35,571	96	218,038
$x^3 – 15x^2 + 500$	100	7187	15,972	118	216,610

Table 31: Optimization performance and timing results

Below we present the figures of each fitness function optimization. In Figure 33, we test 6 different populations for fitness function $F(x) = x$. the algorithm runs for 30 generations. We can observe that the average threshold is generation $g = 22$. After this point no significant increment sum of fitnesses is performed.

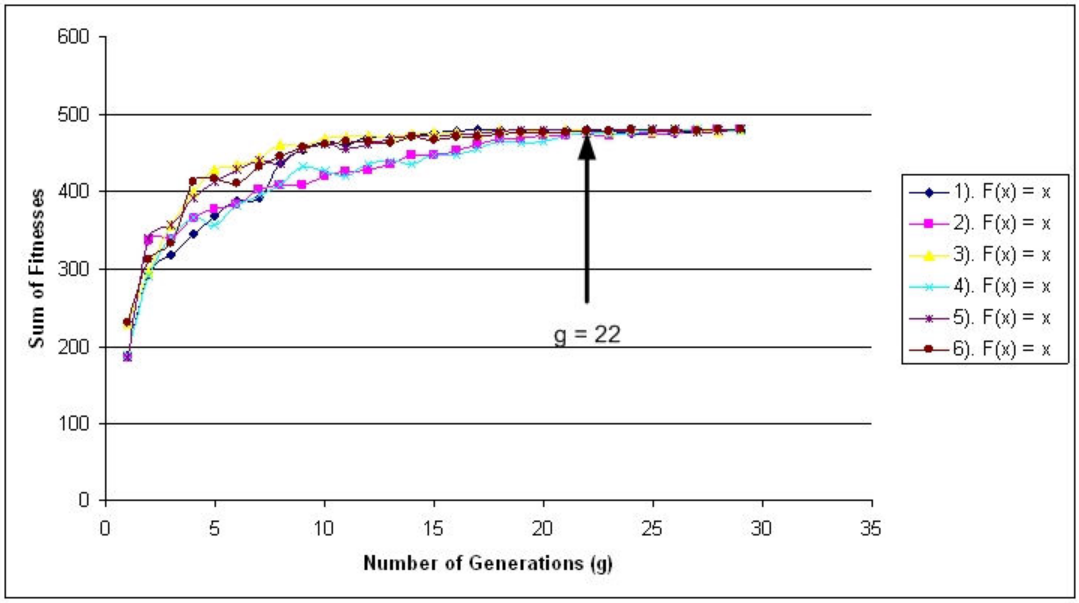

Figure 33: Optimization of fitness function F(x) = x

Figure 34 indicates the optimization of fitness function F(x) = x + 5. Six different populations were tested and the average threshold of the number of generations was 25.

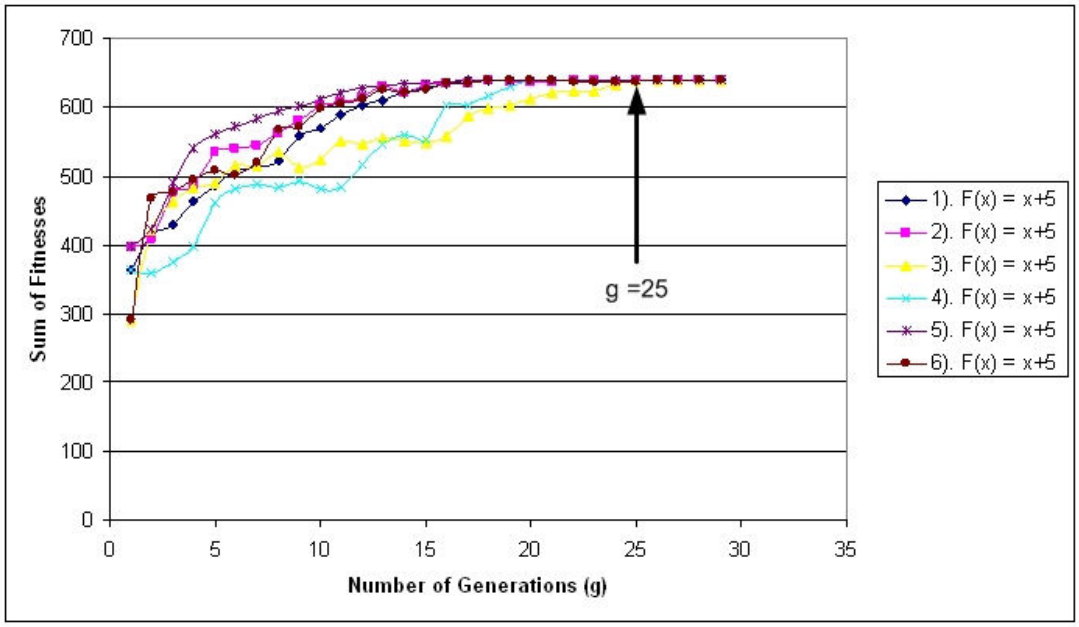

Figure 34: Optimization of fitness function F(x) = x + 5

In Figure 35 fitness function F(x) = 2x was optimized for six populations and the results indicate that after 28 generations the sum of fitnesses has the same value for all the populations.

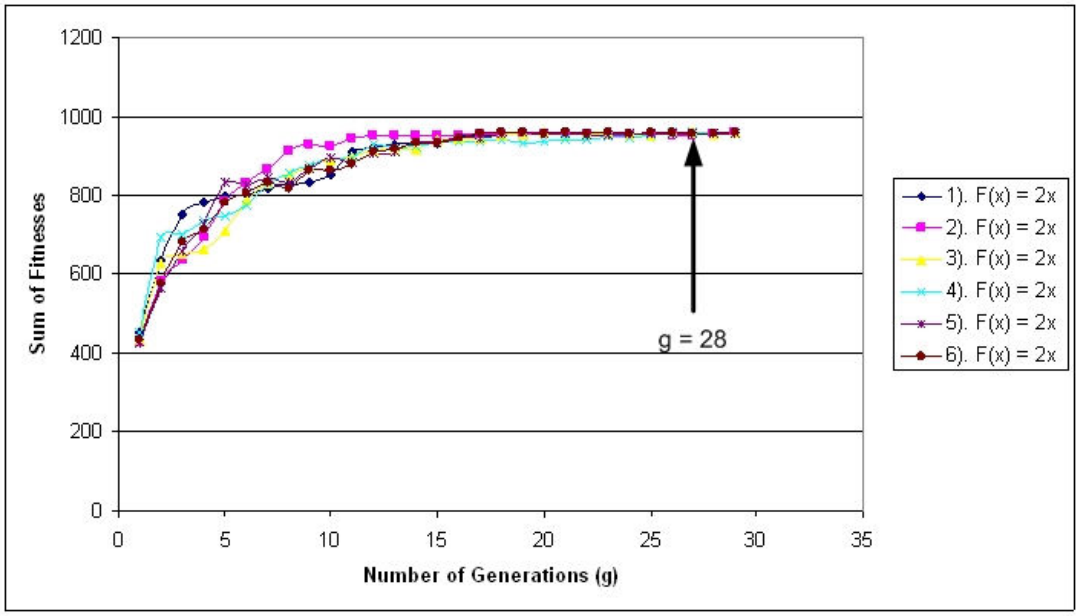

Figure 35: Optimization of fitness function F(x) =2 x

Continuing our optimization analysis of our experiments the next fitness function under examination was $F(x) = x^2$ (Figure 36). The threshold was g = 22 generations.

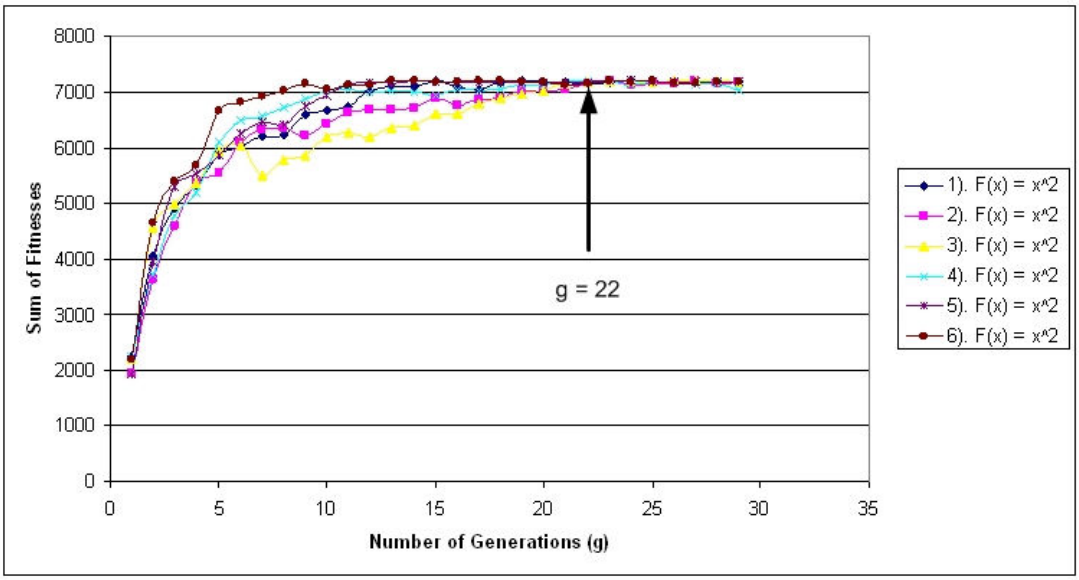

Figure 36: Optimization of fitness function F(x) = x²

The last two complex fitness functions were examined for the optimization process. For the $F(x) = 2x^3 – 45x^2 + 300x$ (Figure 37) we can conclude that need the most number of generations to be optimized (g = 31). In the other hand $F(x) = x^3 – 15x^2 + 500$ (Figure 38) needs only 12 generations to be optimized.

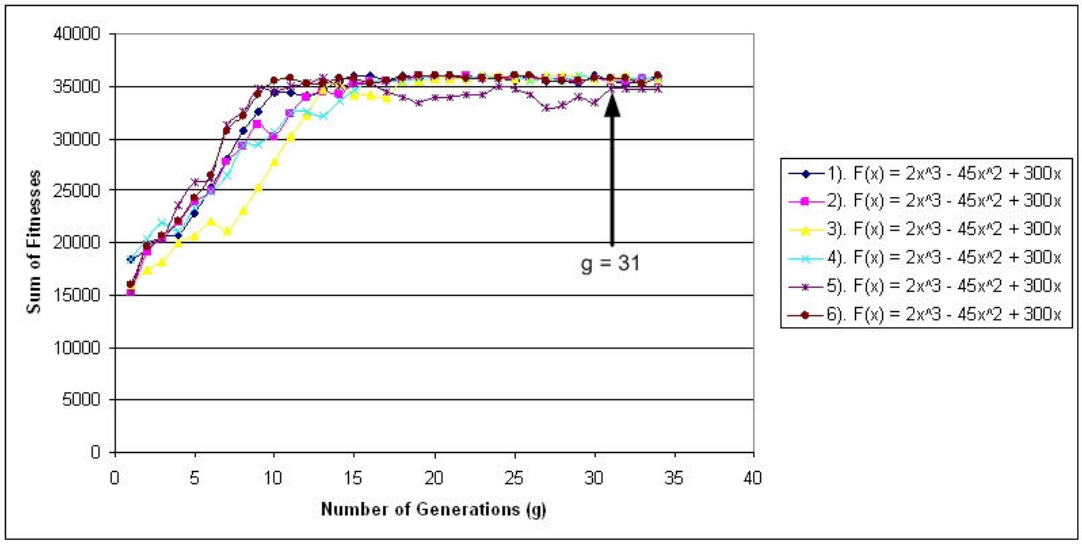

Figure 37: Optimization of fitness function $F(x) = 2x^3 – 45x^2 + 300x$

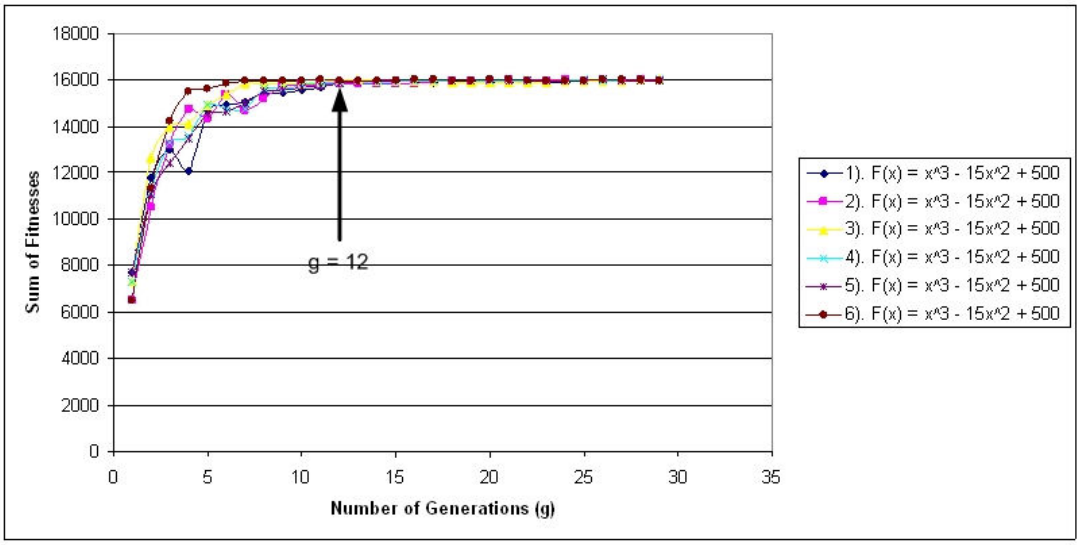

Figure 38: Optimization of fitness function $F(x) = x^3 – 15x^2 + 500$

As we can observe from the Table 31 above all of the fitness functions needs almost the same clock cycles to be optimized for specified number of generations. This happens because we implemented them having one entire control unit so the results are the expected. It is worth noticing that for the HGA runs only one selection module was used. More specific analysis will be done in section 5.2 and 5.3.

5.2. Validation

We didn't found any output file of the Scott implementation in order to validate our results. Two levels of functional verification were used. First each module was individually tested to confirm that it operated correctly under all conceivable conditions. Testbenches were used to confirm the correct functionality of each module and of the overall design. Modelsim 6 SE was used for the simulations we described above. The second level of functional verification involved simulating the HGA on different fitness function in order to see how well the functions were optimized. In all tests, the population was optimized well. In a small number of generations, average fitness increased substantially.

First of all, the VHDL code of the HGA we found at the internet, included only fitness function F(x) = 2x, although Scott supports that more fitness functions were implemented in simulation mode. The member's width was n = 4 bits and the fitness value width of bits, was only f = 5 bits. From simulation analysis we found that fitness function F(x) = 2x was optimized for 20 generations at the same clock cycles as Scott supports in his Master thesis. Moreover the average fitness of the final population was increased. These observations lead us to the validation with Scott implementation. The design was downloaded in a Virtex II Pro FPGA platform and the results were the same as simulation.

Furthermore, because of the small resources of the HGA in the FPGA we inserted more fitness functions to be optimized. In order to support more complex fitness functions the fitness value width was increased to f = 14. Moreover a lot of vectors

were changed inside the VHDL code due to this change. So from the architecture of 9 bits we pass to the 18 bits architecture (n + f = 4 + 14 =18). The fitness evaluation module implemented with DSP's was first tested individually to verify correct functionality. Then we simulate it with the entire design and check the overall's system functionality. The algorithm was first run for the fitness function F(x) = 2x and was compared with the first implementation. We didn't get the same results (the same members at the final population) because of the pipeline of the fitness evaluation module. Moreover the RNG seed of the first implementation was 9 bits and becomes 18 bits. As we previously mentioned RNG is a key component of the system, and its output depends on the RNG seed input. Just like a software-based pseudo random number generator, the RNG module is deterministic so using the same initial RNG seed will yield the same sequence of random numbers. The algorithm was run almost for the same clock cycles but the average fitness of the final population was increased in the same number of generations. This implementation with the DSP's was downloaded for all the fitness functions and the results were the same as the simulation. Here we present Table 32 from Scot's Master Thesis that indicates the correct functionality:

F(x)	Number of Generations (g)	Mean Fitness (Initial Population)	Mean Fitness (Final Population)	Increment Percentage (%)	Clock Cycles
x	15	7.3	14	92	32,429
2x	6	14.62	27.94	91	13,139
x^2	20	74.5	215.2	214	45,019
$2x^3 – 45 x^2 + 300x$	20	572.62	1,049	83	44,503
$x^3 – 15x^2 + 500$	20	248.19	492.25	98	44,317

Table 32: Optimization results from Scott implementation

F(x)	Number of Generations (g)	Mean Fitness (Initial Population)	Mean Fitness (Final Population)	Percentage (%)	Clock Cycles
x	15	6.2	14.43	132	32,002
2x	6	13.6	25.1	84	12,881
x^2	20	66.3	222.1	234	42,529
$2x^3 – 45 x^2 + 300x$	20	516.8	1,111.6	115	43,210
$x^3 – 15x^2 + 500$	20	224.6	498.9	122	44,399

Table 33: Optimization results from our implementation

From Tables 32 and 33 above, we can conclude that average fitness increased by almost the same coefficient for several fitness functions for the same number of generations. Moreover, Scott supports that the actual evaluation time of completion of the algorithm grows linearly with the number of generations. This was realized from us when we conducting the experiments for several number of generations. So, observing Tables 32, 33 we can say that algorithm runs almost at the same clock cycles for several fitness functions.

5.3. Considerations for improvement

The above analysis of the HGA leads us to propose several design improvements:

- Increase parallelization of the selection modules as indicated in Figure 26. At this point we must mention that our implementation supports only one selection module (n_{sel}). Scott supports that inserting two parallel selection modules can reduce the number of clock cycles significant (at the half). Moreover there is a limit (n_{sel} = 3 or 4) where after that, the bottleneck lies on the fitness module and no significant speedup achieved.

- Parallelize the selection-crossover-fitness pipelines, but this would require more complex inter-module communication protocol than the aynchronous handshaking protocol used in this design.

- Make the inter-module communication protocol more efficient. The current handshaking protocol requires four clock cycles per data transfer. If these delays were reduced the entire HGA would run much faster.

- Speed up the selection module polling method used by the crossover and mutation module. Presently the crossover and mutation module polls only one selection module per clock cycle. There would be time to poll several selection modules in a single clock cycle.

- Buffer the outputs of all the modules would reduce the delays associated with some modules waiting for service and blocking others that are waiting upstream in the pipeline.

Chapter 6

Present status and future work

6.1. Present status

The present status of this thesis is that we have an embedded system that implements a function optimization using a genetic algorithm. A lot of functions can be optimized as described in previous chapter. We achieve 50 X speedup than Scott implementation only with porting the design from BORG's board to a Virtex II pro FPGA platform. This is because the clock frequency of the BORG's board was 2 MHz and the Xilinx university platform has a 100 MHz clock. Our implementation supports more fitness functions that are implemented in hardware than Scott implementation which supports only one (F(x) = 2x).

6.2. Future work

The state of the art in FPGA technology will surely advance in the future so improved FPGA technologies could be exploited to improve HGA's capabilities. For example, the genetic parameters of this design could be scaled up so the HGA could handle larger strings, larger populations, more complex fitness functions and more advanced genetic operators. As it analyzed in Chapter 2, the present status of the genetic algorithms implemented in FPGAs indicates that the maximum population size is 100,000 and the maximum number of generations is 1,000,000. The genetic algorithm could be extended by implementing other genetic algorithm operators including multi-point crossover and mutation. Furthermore, HGA could also support alternative encoding schemes such as floating point or signed integer coding.

To increase parallelism multiple selection, crossover and mutation and fitness modules could be added in order to speed up the genetic algorithm. Parallelizing the selection modules will reduce the clock cycles at the half.

We also propose the idea of partial reconfiguration. Probably, not all of the fitness functions should be placed in the FPGA, so we can change the function which is going to be optimized on the fly. User will select onlu the fitness function he wants to optimize and the other fitness functions will not exist in the FPGA. This process could be done by building several bitstreams for fitness function and store them to System Ace Memory. Each time the appropriated bitsream will run for the specified fitness function. This will reduce utilization resources and maybe the speed of the algorithm.

Moreover user could send via PowerPC the coefficients of the polynomial fitness function $ax^3 + bx^2 + cx + d$ he wants to be optimized. Furthermore, the selection of the fitness function that is going to be optimized can be added to the input file so the user doesn't have to re-download the design.

Appendix A

In appendix A we present the inputs and the outputs ports of basic modules of our design.

Signals	Description
INPUTS	
go	Go ahead signal from PowerPC to start the algorithm. If go=1 the algorithm starts
RST	Asynchronous signal from PowerPC to reset the algorithm. If reset=1 the algorithm doesn't run
datain[17:0]	Data arrived from memory and they are forwarded to other modules
toggle	Tells which population to access for fitness and population sequencer
CLK	System clock
reqrng	Request from RNG module
addrrng[2:0]	The address received from RNG which tells the MIM where in the memory is stored the seed for RNG
reqxov	Request from crossover and mutation module
addrxov[2:0]	The address received from crossover and mutation module which tells the MIM where in the memory is stored the address of crossover and mutation probability
reqseq[1:0]	Request from population sequencer module, *reqseq(1)* indicates that population sequencer wants to read a member and *reqseq(0)* that PS wants to read the population size

addrseq[4:0]	The address received from population sequencer module initially tells the MIM where in the memory is stored the address of population size and then *addrseq* tells the MIM the address of member it wants to read
reqfit[1:0]	Request from fitness module, *reqfit(0)* indicates that FM wants to read population size, sum of fitness and number of generations and *reqfit(1)* that FM wants to write to MIM a new member
valfitin[17:0]	Receives a new member from fitness module
fitdone	If this signal is high, this indicates the end of the algorithm and it comes from the fitness module
addrfit[4:0]	Firstly MIM receives addresses of sum of fitnesses, number of generations and population size and then it receives the address of the new member that FM wants to write
OUTPUTS	
done	This signal indicates the end of algorithm and MIM send it to Power Pc
init	It initializes the other modules (PS,FM,CMM,RNG). If RST is high then init is low and conversely
address[6:0]	This signal sent to MEMORY either for reading or writing the address of a member or requesting an HGA run-time parameter
dataout[17:0]	Data of a new member sent to MEMORY which received from FM module
rw	Read or write signal sent to MEMORY (0 is for reading operation and 1 for writing)
toggleout	Shows the final value of toggle sent to Power Pc(0 means that final population is located in positions 6-37 in MEMORY and 1 in positions

	38-69)
ackrng	An acknowledge signal to RNG module
ackxov	An acknowledge signal to Crossover and Mutation module
ackseq	An acknowledge signal to Population Sequencer module
ackfit	An acknowledge signal to Fitness Module
valout[18:0]	MIM passes data to whoever requested them

Table 34: Memory Interface Module

Signals	**Description**
	INPUTS
ackxover	An acknowledge signal from Crossover and Mutation module and if it is 1 then selection module outputs the pair of selected members
sof[18:0]	The sum of fitnesses of the current population
rand1[3:0]	Gets random number from RNG module to scale down the sum of fitness
rand2[3:0]	Gets random number from RNG module to scale down the sum of fitness
reset	Fitness module resets the Selection module when the current generation has ended and the populations have switched
CLK	System clock
dup	This signal is set to '1' when we have the same input as previous

input[17:0]	A new member received from Population Sequencer
OUTPUTS	
aout[3:0]	The first member selected and send it to Crossover and Mutation module
bout[3:0]	The second member selected and send it to Crossover and Mutation module
reqxover	Sends a request to Crossover and Mutation module when the pair of members is ready. This signal is set high randomly.

Table 35: Selection Module

Signals	**Description**
INPUTS	
init	An initiallizing signal send from MIM
CLK	System clock
ackmem	Acknowledge signal from MIM
param[15:0]	Value of RNG seed sent from MIM
OUTPUTS	
reqmem	A request sent to MIM
domut[3:0]	Random bit string sent to Crossover and Mutation module for determining whether to perform mutation
doxover[3:0]	Random bit string sent to Crossover and Mutation module for determining whether to perform crossover

mutpt[1:0]	Shows what the mutation point is
xoverpt[1:0]	Shows what the crossover point is
addr[2:0]	Send RNG seed address
randsel1[3:0]	Random bit string send to the Selection Module for scaling down the sum of fitness
randsel2[3:0]	Random bit string send to the Selection Module for scaling down the sum of fitness

Table 36: Random Number Generation

Signals	**Description**
	INPUTS
init	An initializing signal send from MIM
CLK	System clock
domut[3:0]	Random bit string received from RNG module for determining whether to perform mutation
doxover[3:0]	Random bit string received from RNG module for determining whether to perform crossover
mutpt[1:0]	Shows what the mutation point is
xoverpt[1:0]	Shows what the crossover point is
ain[3:0]	The first member received from Selection module
bin[3:0]	The second member received from Selection module

reqsel	Bus request from Selection module
ackfit	Bus acknowledge from Fitness module
ackmem	Bus acknowledge from MIM
param[3:0]	The crossover and the mutation probabilities received from MIM
OUTPUTS	
acksel	Bus acknowledge to Selection module
reqfit	Bus request to Fitness module
aout[3:0]	First member to fitness module
bout[3:0]	Second member to fitness module
reqmem	Bus request to MIM
addr[3:0]	Address of initial parameters(crossover and mutation probabilities)

Table 37: Crossover and Mutation Module

Signals	**Description**
INPUTS	
wea	This signal is for the write operation in A port of the Memory. It is high when we want to write something to Memory and low for reading operation
clka	System clock at port A
clkb	System clock at port B

dina[17:0]	Data stored into port A of Memory
dinb[17:0]	Data stored into port B of Memory
web	This signal is for the write operation in B port of the Memory. It is high when we want to write something to Memory and low for reading operation
ena	If this signal is high then read or write operations performed in port A
enb	If this signal is high then read or write operations performed in port B
addra[6:0]	The address of the data we want to write to port A of Memory
addrb[6:0]	The address of the data we want to write to port B of Memory
OUTPUTS	
douta[17:0]	Memory outputs data to other modules from port A
doutb[17:0]	Memory outputs data to other modules from port B

Table 38: Memory Module

Signals	**Description**
INPUTS	
clock	System clock
rst	System reset
x[3:0]	The member sent from fitness module for evaluation
init_mult_ctrl	Input from fitness module which enables first fitness evaluation module

selfitfunc[2:0]	Select one of the six functions to be optimized
OUTPUTS	
output[13:0]	The output of the address that becomes input for Memory module
resultrdy	Informs fitness module that fitness value from first fitness evaluation module is ready

Table 39: Fitness Evaluation Module

Signals	**Description**
INPUTS	
init	An initializing signal send from MIM
CLK	System clock
value[17:0]	This signal has the population size sent from Memory Interface Module
ackmem	If this signal is set means that Population sequencer got Memory acknowledge
OUTPUTS	
dup	If this signal is high means that the same member as last one was passed to selection module
reqmem[1:0]	Request sent to Memory Interface Module. If reqmem (0)=1 it request memory access and if reqmem(0)=0, tells Memory Interface Module that address sent. If reqmem(1)=1, prepares to write new members to memory
addr[4:0]	This is sent to memory interface module for request data from

	specified address
output[17:0]	The member passed to selection module

Table 40: Population Sequencer

Signals	Description
INPUTS	
init	An initializing signal send from MIM
CLK	System clock
reqxover	Bus request from Crossover and Mutation module
toggleinit	A signal shows in which area of memory is placed the initial population (0 means that initial population is located in positions 6-37 in MEMORY and 1 in positions 38-69)
ackmem	If this signal is set it indicates that Fitness Module got memory acknowledge
param[18:0]	The sum of fitnesses, population size and number of generations of current population send from Memory Interface Module
ain[3:0]	The first member received Crossover and Mutation module
bin[3:0]	The second member received from Crossover and Mutation module
offchipfitresa[13:0]	The fitness value of the first member calculated by the Fitness Evaluator
offchipfitresb[13:0]	The fitness value of the second member calculated by the Fitness Evaluator

resultrdy1	It is set to high indicating that the result from first fitness evaluation module has evaluated the fitness value from the first member.
resultrdy2	It is set to high indicating that the result from second fitness evaluation module has evaluated the fitness value from the second member.
OUTPUTS	
done	This signal tells the Memory Interface Module to shut down
ackxover	Acknowledge the Crossover and Mutation Module
reset	If this signal is high, resets the Selection Module
sof[18:0]	Sum of fitnesses of the current population sent to Selection Module
toggle	This signal is sent to Memory Interface Module to inform it which is the current population
reqmem [1:0]	Request sent to Memory Interface Module. If reqmem (0) =1 it request memory access and if reqmem (0)=0, tells Memory Interface Module that address sent. If reqmem(1)=1, prepares to write new members to memory
addrmem [4:0]	Send address of the population size, the number of generations, sum of fitnesses and also the address of the new member to be written in Memory
newmember [17:0]	The new member sent to Memory Interface Module to be written in Memory
init_mult_ctrl1	Enables the first fitness evaluation module
init_mult_ctrl2	Enables the second fitness evaluation module
writemem	Write enable signal for the sum of fitnesses memory

memaddr[6:0]	Send address to sum of fitnesses memory
mem_sum_enb	Enables sum of fitnesses memory for read and write operation

Table 41: Fitness Module

Signals	**Description**
	INPUTS
wea	This signal is for the write operation in A port of the Sum of Fitnesses Memory. It is high when PowerPC writes something to Sum of Fitnesses Memory and low for reading operation
clka	System clock at port A
clkb	System clock at port B
dina[18:0]	Data stored into port A of Sum of Fitnesses Memory.
dinb[18:0]	Data stored into port B of Sum of Fitnesses Memory.
web	This signal is for the write operation in B port of the Sum of Fitnesses Memory. It is high when we want to write something to Sum of Fitnesses Memory and low for reading operation
ena	Only if this signal is high then read or write operations performed in port A
enb	Only if this signal is high then read or write operations performed in port B
addra[6:0]	The address of the data we want to write to port A of Sum of Fitnesses Memory
addrb[6:0]	The address of the data we want to write to port B of Sum of

	Fitnesses Memory
OUTPUTS	
douta[18:0]	Sum of Fitnesses Memory outputs data to other modules from port A
doutb[18:0]	Sum of Fitnesses Memory outputs data to other modules from port B

Table 42: Sum of Fitnesses Memory

Signals	**Description**
INPUTS	
I0[13:0]	The fitness value of the first fitness function
I1[13:0]	The fitness value of the second fitness function
I2[13:0]	The fitness value of the third fitness function
I3[13:0]	The fitness value of the fourth fitness function
I4[13:0]	The fitness value of the fifth fitness function
I5[13:0]	The fitness value of the sixth fitness function
S[2:0]	The selection for the multiplexer is the fitness function choice (selfitfunc)
OUTPUTS	
O[13:0]	Outputs one of the inputs depending on the selection signal

Table 43: Multiplexer

Signals	Description
INPUTS	
clk	System clock
rst	System reset
rdy1	Result ready from first multiplier
rdy2	Result ready from second multiplier
rdy3	Result ready from third multiplier
rdy4	Result ready from fourth multiplier
rdy5	Result ready from fifth multiplier
init_mult_ctrl	Initialize signal from fitness module to functional controller module
OUTPUTS	
nd1	New data to first multiplier
sclr1	Synchronous clear to first multiplier
sclr2	Synchronous clear to second multiplier
sclr3	Synchronous clear to third multiplier
sclr4	Synchronous clear to fourth multiplier
sclr5	Synchronous clear to fifth multiplier
resultrdy	Output to fitness module to inform that result is ready

Table 44: Functional Controller

We don't present the second fitness evaluation module because it has the same structure.

Signals	Description
INPUTS	
I[6:0]	The address for memory which is sent from the PowerPC
clk	System clock
load	This signal is set to high to enable register
Q[6:0]	The address from the input is outputted after a clock cycle if load is high
OUTPUTS	
Q[6:0]	The address from the input is outputted after a clock cycle if load is high

Table 45: Register

The second register is not presented here because has the same structure.

Signals	Description
INPUTS	
x[6:0]	The address for data memory which is sent from the PowerPC
y[6:0]	The address for sum of fitnesses memory which is sent from the PowerPC

clk	System clock
done	This signal indicated the end of the algorithm when is high
OUTPUTS	
PowerPC_ena_out	This signal is high for one clock cycle, when we have change on the address sent from PowerPC to enable port A of the data memory.
PowerPC_sum_ena	This signal is high for one clock cycle, when we have change on the address sent from PowerPC to enable port A of the sum of fitnesses memory.
hag_enb_out	This signal is high while done signal is set to low, to enable port B of the data memory.

Table 46: Synchronization controller

Appendix B

In this appendix a detailed block diagram with all the signals of the HGA is presented (Figure 39). Due to restricted area some modules and their signals are omitted.

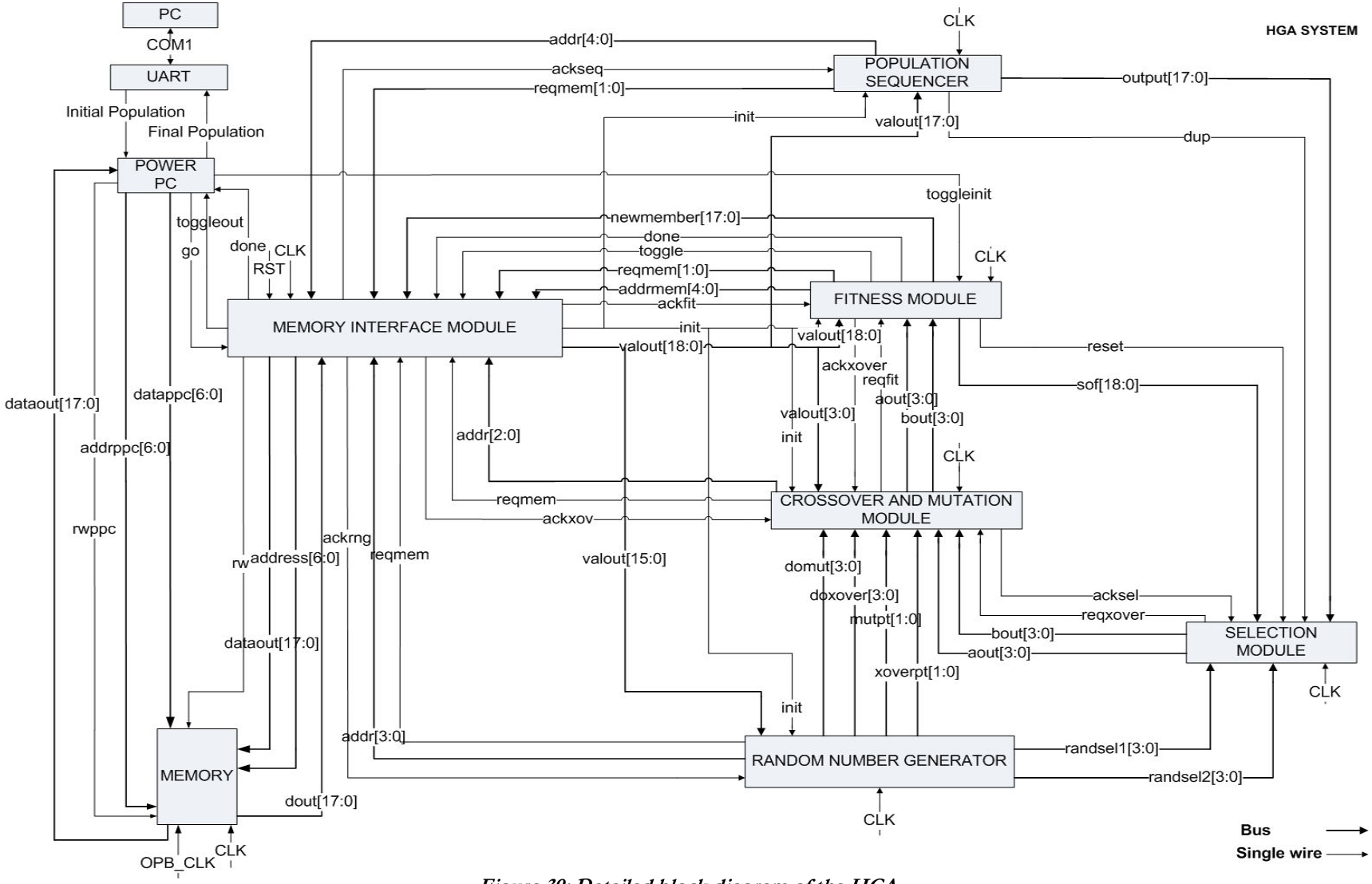

Figure 39: Detailed block diagram of the HGA

References

[1] Peter Martin, "A Hardware Implementation of a Genetic Programming System Using FPGAs and Handel-C", 2001.

[2] Chatchawit Aporntewan and Prabhas Chongstitvatana, "A Hardware Implementation of the Compact Genetic Algorithm". Proceedings of the 2001 congress on Evolutionary Computation Seoul, Korea.

[3] G. Koonar, S. Areibi, M. Moussa, "Hardware Implementation of Genetic Algorithms for VLSI CAD Design".

[4] Wallace Tang and Leslie Yip, "Hardware Implementation of Genetic Algorithms Using FPGA". The 47th IEEE International Midwest Symposium on Circuits and Systems, IEEE 2004.

[5] Heywood M.I. Zincir-Heywood A.N, "Register Based Genetic Programming on FPGA Computing Platforms", Appears in EuroGP 2000.

[6] Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto and Minoru Ito, "A Hardware Implementation Method of Multi-Objective Genetic Algorithms".

[7] John R. Koza, Stephen L. Bade, Forrest H Bennett III, Martin A. Keane, Jeffrey L. Hutchings and David Andre, "Evolving Computer Programs using Rapidly Reconfigurable Field-Programmable Gate Arrays and Genetic Programming".

[8] Matti Tommiska and Jarkko Vuori, "Hardware Implementation of GA", Finland August 2006.

[9] Hossam E. Mostafa, Ahmed I. Khadragi and Yasser Y. Hanafi, "Hardware Implementation of Genetic Algorithm on FPGA", 21 National Radio conference March 2004.

[10] Stephen D. Scott, Ashok Samal and Sharad Seth, "HGA: A Hardware-Based Genetic Algorithm". In the Proceedings of ACM/SIGDA Third International Symposium on Field Programmable Gate Arrays 1995, pp.53-59.

[11] H. Emam, H. Fekry, M. A. Ashour and A. M. Wahdan, "Introducing an FPGA based - genetic algorithms in the applications of blind signal separation". The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications 2003 IEEE.

[12] Iouliia Skliarova and Anto´nio de Brito Ferrari, "Reconfigurable Hardware SAT Solvers: A Survey of Systems".

[13] Simon Perkins, Reid Porter, and Neal Harvey, "Everything on the Chip: A Hardware-Based Self-Contained Spatially-Structured Genetic Algorithm for Signal Processing".

[14] Tatsuhiro Tachibana, Yoshihiro Murata, Naoki Shibata, Keiichi Yasumoto and Minoru Ito, "General Architecture for Hardware Implementation of Genetic Algorithm". 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06).

[15] Tu Lei and Zhu Ming-cheng, Wang Jing-xia, "The Hardware Implementation of a Genetic Algorithm Model with FPGA", WO02 IEEE.

[16] Man F. So and Angus Wu, "Hardware implementation of four-step genetic search algorithm".

[17] Paul Graham and Brent Nelson, "A Hardware genetic algorithm for the traveling salesman problem on SPLASH 2".

[18] Kyrre Glette and Jim Torresen and Moritoshi Yasunaga, "Online Evolution for a High-Speed Image Recognition System Implemented On a Virtex-II Pro FPGA". Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007) 2007 IEEE.

[19] Barry Shackleford, Greg Snider, Richard J. Carter, "A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine". 2001 Kluwer Academic Publishers. Manufactured in The Netherlands.

[20] Stephen D. Scott, Ashok Samal and Sharad Seth, "A synthesizable VHDL coding of a genetic algorithm". University of Nebraska Lincoln November 1997.

[21] Xilinx University Program Virtex II Pro Development System March, 2005.

[22] Virtex II Pro and Virtex II Pro X Platform FPGAs: Complete datasheet October, 2005, ds083.

[23] Stephen D. Scott, Ashok Samal and Sharad Seth, "Hardware based genetic algorithm". Master thesis at University of Nebraska Lincoln August 1994.

[24] http://www.hurgh.org/ascbinhex.php.

[25] http://en.wikipedia.org/wiki/Genetic_algorithm.

[26] http://cse.unl.edu/~sscott/research/r-area.shtml.

[27] K. A. De Jong and W. M. Spears, "Using genetic algorithms to solve NP-complete problems". Proceedings of the Third International Conference on Genetic Algorithms, pp 124-132, June 1989.

[28] Micaela Serra, Terry Slater, "The analysis of one-dimensional linear cellular automata and their aliasing properties". IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp767-778, July 1990.

[29] Stephan Wolfram, "University and complexity in cellular automata", Physica, pp 1-35, 1984.