Chapter 1
A Reconfigurable Hardware for Genetic Algorithms

Abstract. In this chapter, we propose a massively parallel architecture of a hardware implementation of genetic algorithms. This design is quite innovative as it provides a viable solution to the fitness computation problem, which depends heavily on the problem-specific knowledge. The proposed architecture is completely independent of such specifics. It implements the fitness computation using a neural network. The hardware implementation of the used neural network is stochastic and thus minimises the required hardware area without much increase in response time. Last but not least, we demonstrate the characteristics of the proposed hardware and compare it to existing ones.

1.1 Introduction

Generally speaking, a *genetic algorithm* is a process that evolves a set of *individuals*, also called *chromosomes*, which constitutes the *generational population*, producing a new population. The individuals represent a solution to the problem in consideration. The freshly produced population is yield using some genetic operators such as *selection*, *crossover* and *mutation* that attempt to simulate the natural breeding process in the hope of generating new solutions that are *fitter*, i.e. adhere more the problem constraints.

Previous work on hardware genetic algorithms can be found in [5, 10, 12]. Mainly, Earlier designs are hardware/software codesigns and they can be divided into three distinct categories: *(i)* those that implement the fitness computation in hardware and all the remaining steps including the genetic operators in software, claiming that the bulk computation within genetic evolution is the fitness computation. The hardware is problem-dependent; *(ii)* and those that implement the fitness computation in software and the rest in hardware, claiming that the ideal candidate are the genetic operators as these exhibit regularity and generality [2, 7]. *(iii)* those that implement the whole genetic algorithm in hardware [10]. We believe that both approaches are worthwhile but a hardware-only implementation of both the fitness calculation and genetic operators is also valuable. Furthermore, a hardware implementation that is problem-independent is yet more useful.

N. Nedjah and L. de Macedo Mourelle, *Hardware for Soft Computing and Soft Computing for Hardware*, Studies in Computational Intelligence 529,
DOI: 10.1007/978-3-319-03110-1_1,

The remainder of this chapter is divided into five sections. In Section 1.2, we describe the principles of genetic algorithms. Subsequently, in Section 1.3, we propose and describe the overall hardware architecture of the problem-independent genetic algorithm. Thereafter, in Section 1.4, we detail the architecture of each of the component included in the hardware genetic algorithm proposed. Then, in Section 1.5, assess the performance of the proposed architecture. Finally, we draw some conclusions 9.9.

1.2 Principles of Genetic Algorithms

Genetic algorithms maintain a *population* of *individuals* that evolve according to *selection rules* and other *genetic operators*, such as *mutation* and *crossover*. Each individual receives a measure of *fitness*. Selection focuses on high fitness individuals. Mutation and crossover provide general heuristics that simulate the reproduction process. Those operators attempt to perturb the characteristics of the parent individuals as to generate distinct offspring individuals.

Genetic algorithms are implemented through the procedure described by Algorithm 1.1, wherein parameters *ps*, *ef* and *gn* are the population size, the expected fitness of the returned solution and the maximum number of generation allowed respectively.

Algorithm 1.1. GA – Genetic algorithms basic cycle

Require: population size *ps*, expected fitness *ef*, generation number *gn*
Ensure: the problem solution
  generation := 0
  population := initialPopulation()
  fitness := evaluate(population)
  repeat
    parents := select(population)
    population := mutate(crossover(parents))
    fitness := evaluate(population)
    generation := generation + 1
  until (fitness[i] = *ef*, $1 \leq i \leq ps$) OR (generation $\geq gn$)

In Algorithm 1.1, function *intialPopulation* returns a valid random set of individuals that compose the population of first generation while function *evaluate* returns the fitness of a given population storing the result into fitness. Function *select* chooses according to some random criterion that privilege fitter individuals, the individuals that should be used to generate the population of the next generation and function *crossover* and *mutate* implement the crossover and mutation process respectively to actually yield the new population.

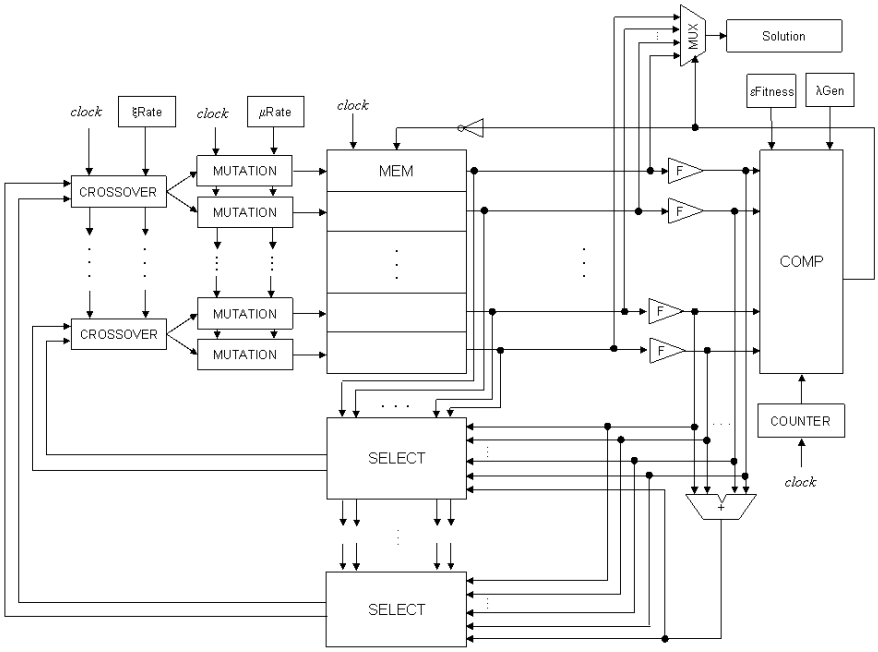


Fig. 1.1 Overall architecture of the hardware genetic algorithm proposed

1.3 Overall Architecture for the Hardware Genetic Algorithm

Clearly, for hardware genetic algorithms, individuals are always represented using their binary representation. Almost all aspects of genetic algorithms are very attractive for hardware implementation. The selection, crossover and mutation processes are generic and so are problem-independent. The main issue in the hardware implementation of genetic algorithms is the computation of individual's fitness values. This computation depends on problem-specific knowledge. The novel contribution of the work consists of using neural network hardware to compute the fitness of individuals. The software version of the neural network is trained with a variety of individual examples. Using a hardware neural network to compute individual fitness yields a hardware genetic algorithm that is fully problem-independent.

The overall architecture of the proposed hardware is given Fig. 1.1. It is massively parallel. The selection process is performed in one clock cycle while the crossover and mutation processes are completed within two clock cycles.

The fitness of individual in the generational population is evaluated using hardware neural networks, which take advantage of stochastic representation of signals to reduce the hardware area required [9]. Stochastic computing principles are well

detailed in [4]. The motivation behind the use of stochastic arithmetic is its simplicity. Designers are faced with hardware implementations that are very large due to large digital multipliers, adders, etc.. Stochastic arithmetic provides a way of performing complex computations with very simple hardware. Stochastic arithmetic provides a very low computation hardware area and fault tolerance. Adders and subtracters can be implemented by an ensemble of multiplexers and multipliers by a series of XOR gates. (For formal proofs on stochastic arithmetic, see [3, 9].)

1.4 Detailed Component Architectures

In this section, we concentrate on the hardware architecture of the components included in the overall architecture of Fig. 1.1.

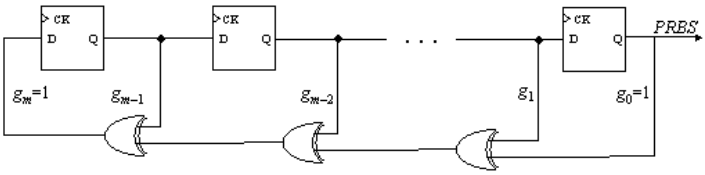
1.4.1 Shared Memory for Generational Population

The generational population is kept in a synchronised bank of registers that can be read and updated. The basic element of the shared memory is an individual. An individual is a simply a bit stream of fixed size. The memory is initially filled up with a fixed initial population. Each time the comparator's output (i.e. component COMP) is 0, the content of the registers changes with the individuals provided as inputs. This happens whenever the fitness f of the best individual of the current generation is not as expected (i.e., $f > \varepsilon Fitness$) and the current generation g is not the last allowed one (i.e., $g \neq \lambda Gen$). Note that $\varepsilon Fitness$ and λGen are two registers that store the expected fitness value and the maximum number of generation allowed respectively.

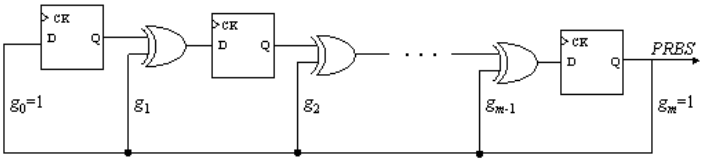
1.4.2 Random Number Generator

A central component to the proposed hardware architecture of genetic algorithms is a source of pseudorandom noise. A source of pseudorandom digital noise consists of a *linear feedback shift register* or LFSR, described by first in [1] and by many others, for instance [3], LFSRs are very practical as they can easily be constructed using standard digital components.

Linear feedback shift registers can be implemented in two ways. The Fibonacci implementation consists of a simple shift register in which a binary-weighted modulo-2 sum of the taps is fed back to the input. Recall that modulo-2 sum of two one-bit binary numbers yields 0 if the two numbers are identical and 1 if not. The Galois implementation consists of a shift register, the content of which is modified at every step by a binary-weighted value of the output stage. The architecture of the LFSR using these methods are shown in Fig. 1.2.



(a) Fibonacci



(b) Galois

Fig. 1.2 Pseudorandom bitstream generators - Fibonacci vs. Galois implementation

Left feedback shift registers such as those of Fig. 1.2 can be used to generate multiple pseudorandom bit sequences. However, the taps from which these sequences are yield as well as the length of the LFSR must be carefully chosen. (See [3, 4] for possible length/tap position choices).

1.4.3 Selection Component

The selection component implements a variation of the roulette wheel selection. The interface of this component consists of all the individuals, say i_1, i_2, ..., i_{n-1}, i_n of the generational population of size n together with the respective fitness, say f_1, f_2, ..., f_{n-1}, f_n and the overall sum of all these fitness values, say *sum*. The component proceeds as described in the following steps:

1. A random number, say ρ is generated;
2. The sum of the individual's fitness values is scaled down using ρ, i.e $ssum := sum - \rho$;
3. Choose an individual, say i_j from the selection pool and cumulate the corresponding fitness f_j, i.e. $csum := csum + f_j$;
4. Compare the scaled sum and the so far cumulated sum and select individual i_j if $csum > ssum$, otherwise go back to step 1;
5. When the first individual is selected, go back to step 1 and apply the same process to select the second individual.

The architecture of the selection component is shown in Fig. 1.3. The above iterative process is implemented using a state machine (CONTROLLER in Fig. 1.3). The state machine has 6 states and the associated state transition function is described in Fig. 1.4. The actions performed in each state of the controller machine are described

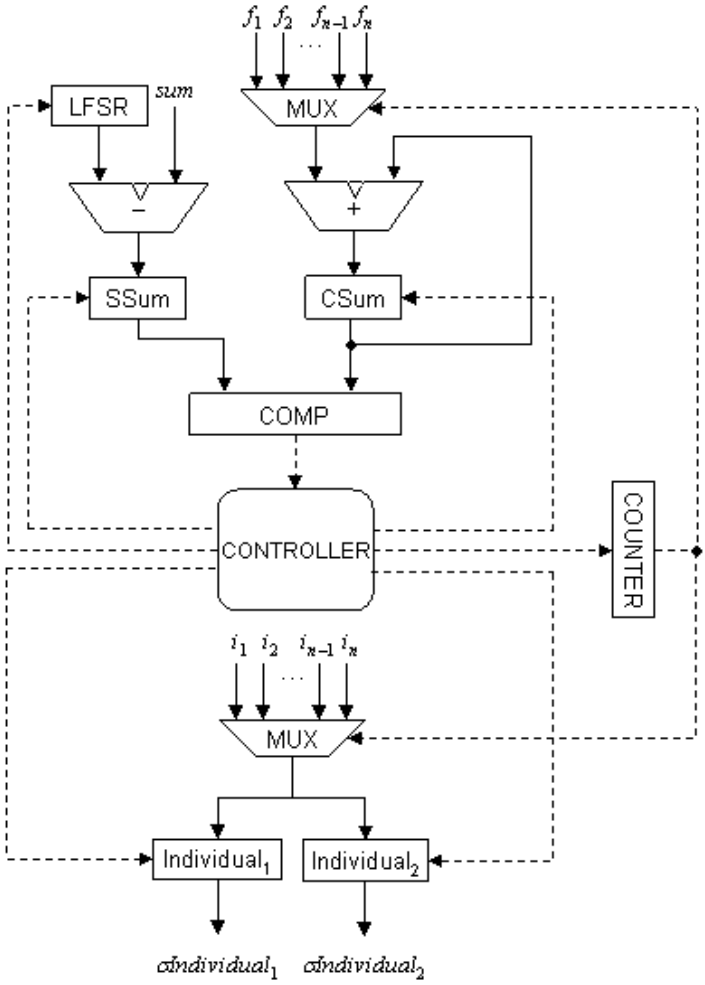


Fig. 1.3 The architecture of the selection component

below. Signal *compare* is set when the either an individual having the expected fitness is found or the last generation has passed.

S_0**:** initialise counter;
load register CSUM with 0;
S_1**:** stop the random number generator;
S_2**:** load register SSUM;
S_3**:** load register CSUM;
S_4**:** if *compare* = 1 then
if *step* = 0 then
load register INDIVIDUAL$_1$;

start the random number generator;
else load register INDIVIDUAL$_2$;
increment the counter;

S_5**:** if $step = 0$ then set $step$;

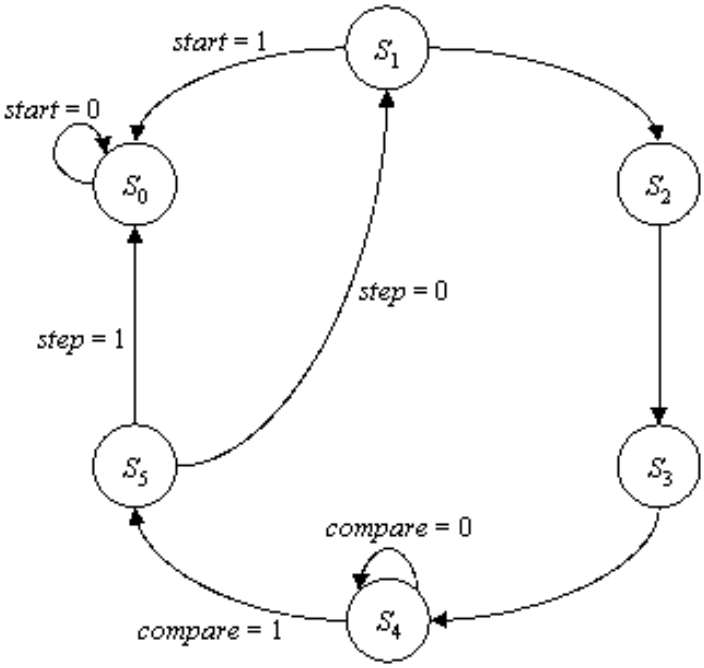


Fig. 1.4 The state transition function of the selection component controller

1.4.4 Genetic Operator's Components

The genetic operators are the crossover followed by the mutation. The crossover component implements the double-point crossover. It uses a linear feedback shift register which provides the random number that allows the component to decide whether to actually perform the crossover or not. This depends on whether the randomised number surpasses the informed crossover rate $\xi Rate$. In the case it does, the bits of the less significant half of the randomised number is used as the first crossover point and the most significant part as the second one.

The mutation component also uses a random number generator. The generated number must be bigger that the given mutation rate $\mu Rate$ for the mutation to occur. The bits of the randomised number are also used as way to choose the mutation

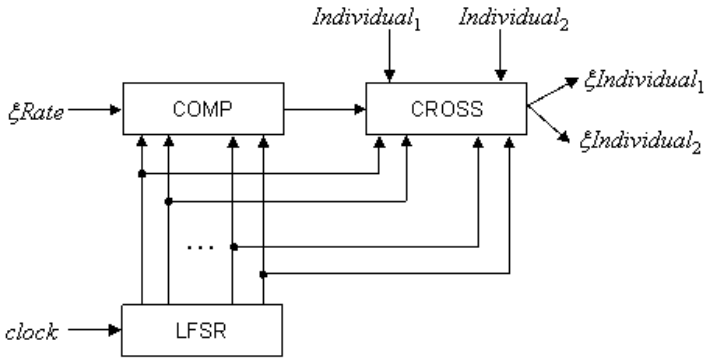


Fig. 1.5 The architecture of the crossover component

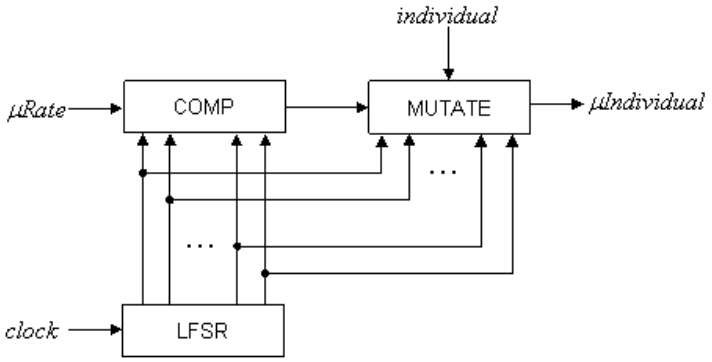


Fig. 1.6 The architecture of the crossover component

degree of the individual. Starting from the less significant bit of both the random number and the individual, if the bit in the former is 1 then the corresponding bit in the later is complemented and otherwise it is kept unchanged. The hardware architecture of the mutation component is given in Fig. 1.6.

1.4.5 Fitness Evaluation Component

The individual fitness measure is estimated using neural networks. In previous work, the authors proposed and implemented a hardware for neural networks [9]. The implementation uses stochastic signals and therefore reduces very significantly the hardware area required for the network. The network topology used is the fully-connected feed-forward. The neuron architecture is given in Fig. 1.7. (More details can be found in [9].) For the hardware genetic implementation, the number of input neurons is the same as the size of the individual. The output neuron are augmented with a shift register to store the final result. The training phase is supposed to be performed before the first use within the hardware genetic algorithm.

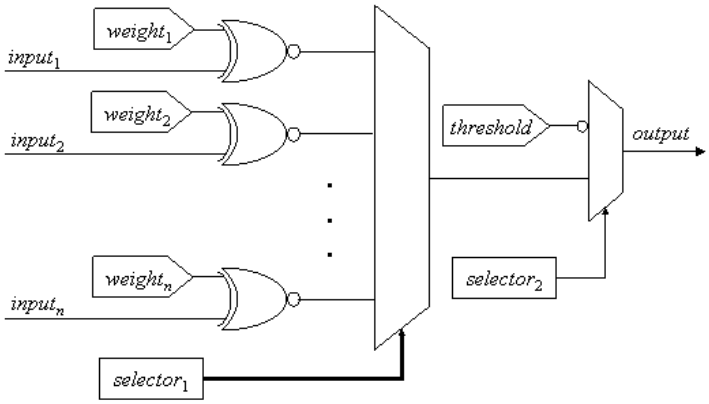


Fig. 1.7 Stochastic bipolar neuron architecture ([9])

1.5 Performance Results

The hardware genetic algorithm proposed was simulated then programmed into an Spartan3 Xilinx FPGA [14]. In order to assess the performance of the proposed hardware genetic algorithm, we maximise the function that was first used in [8]. It was also used by Scott, Seth and Samal to evaluate their hardware implementation for genetic algorithms [11]. The function is not easy to maximise, which is clear from the function plot of Fig. 1.8. The training phase of the neural network was done by software using toolbox offered in MatLab [6].

$$\begin{aligned} & f(x,y) = 21.5 + x\sin(4\pi x) + y\sin(20\pi y), \\ & \\ & -3.0 \leq x_1 \leq 12.1 \\ & 4.1 \leq x_2 \leq 5.8 \end{aligned} \tag{1.1}$$

The characteristics of the software and hardware implementations proposed in [11] and those of the hardware genetic algorithm we proposed in this chapter are compared in Table 1.1. It is clear that the hardware implementations are both much faster than the software version. One can clearly note that our implementation (PHGA) requires more than twice that required by Scott, Seth and Samal's implementation (HEGA). Note, however, that the hardware area necessary to the computation of the fitness function is not included as it is not given in [11]. From another perspective, PHGA is more than five time faster as it is massively parallel. We also believe that the computation of the fitness function is much faster with the neural network. Observe that PHGA evolved a better solution.

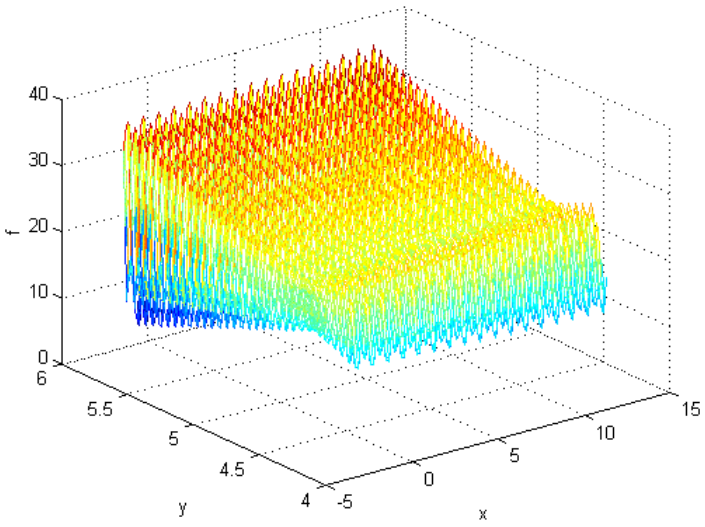


Fig. 1.8 Plotting Michalewics's function ([8])

Table 1.1 Comparison of the performance results: software genetic algorithms (SGA), hardware engine for genetic algorithms (HEGA) and proposed hardware genetic algorithms (PHGA). (The area is expressed in terms of CLBs and the time is in seconds.)

Implementation	time	area	solution	x	y	area× time
SGA	40600	0	38.5764	–	–	–
HEGA	972	870	38.8419	–	–	845640
PHGA	189	1884	38.8483	11.6241	5.7252	356076

1.6 Summary

In this chapter, we proposed a novel hardware architecture for genetic algorithms. It is novel in the sense that is massively parallel and problem-independent. It uses neural networks to compute the fitness measure. Of course, for each type of problem, the neuron weights need to be updated with those obtained in the training phase. Without any doubts, the proposed hardware is extremely faster than the software implementation. Furthermore, it is much faster than the hardware engine proposed by Scott, Seth and Samal in [11]. However, it seems that our implementation requires almost twice the hardware are used to implement their architecture. Nevertheless, we do not have an exact record of the hardware are consumed in [11] as the authors did not provide nor include the hardware required to implement the fitness module for Michalewics's function [8].

References

1. Bade, S.L., Hutchings, B.L.: FPGA-Based Stochastic Neural Networks - Implementation. In: IEEE Workshop on FPGAs for Custom Computing Machines, Napa CA, April 10-13, pp. 189–198 (1994)
2. Bland, I.M., Megson, G.M.: Implementing a generic systolic array for genetic algorithms. In: Proc. 1st. On-Line Workshop on Soft Computing, pp. 268–273 (1996)
3. Brown, B.D., Card, H.C.: Stochastic Neural Computation I: Computational Elements. IEEE Transactions on Computers 50(9), 891–905 (2001)
4. Gaines, B.R.: Stochastic Computing Systems. Advances in Information Systems Science (2), 37–172 (1969)
5. Liu, J.: A general purpose hardware implementation of genetic algorithms, MSc. Thesis, University of North Carolina (1993)
6. MathWorks (2004), `http://www.mathworks.com/`
7. Megson, G.M., Bland, I.M.: Synthesis of a systolic array genetic algorithm. In: Proc. 12th. International Parallel Processing Symposium, pp. 316–320 (1998)
8. Michalewics, Z.: Genetic algorithms + data structures = evolution programs, 2nd edn. Springer, Berlin (1994)
9. Nedjah, N., Mourelle, L.M.: Reconfigurable Hardware Architecture for Compact and Efficient Stochastic Neuron. In: Mira, J., Álvarez, J.R. (eds.) IWANN 2003. LNCS, vol. 2687, pp. 17–24. Springer, Heidelberg (2003)
10. Scott, S.D., Samal, A., Seth, S.: HGA: a hardware-based genetic algorithm. In: Proc. ACM/SIGDA 3rd International Symposium in Field-Programmable Gate Array, pp. 53–59 (1995)
11. Scott, S.D., Seth, S., Samal, A.: A hardware engine for genetic algorithms. Technical Report, UNL-CSE-97-001, University of Nebraska-Lincoln (July 1997)
12. Turton, B.H., Arslan, T.: A parallel genetic VLSI architecture for combinatorial real-time applications – disc scheduling. In: Proc. IEE/IEEE International Conference on Genetic Algorithms in Engineering Systems, pp. 88–93 (1994)
13. Xilinx (2004), `http://www.xilinx.com/`