A Flexible Genetic Algorithm Chip

Shian-De Chen, Pei-Yin Chen and Yung-Ming Wang
Department of Electronic Engineering
Southern Taiwan Univ. of Technology
Tainan, Taiwan 710, R. O. C.
tedvip@yahoo.com.tw, pychen@mail.stut.edu.tw and wym1976@giga.net.tw

Abstract

The genetic algorithm (GA) can find an optimal solution in many complex problems. Therefore, it has been used widely in many applications. A flexible VLSI genetic algorithm processor is proposed in this paper. It can perform dynamically various fitness functions, four crossover operations, and over ten thousand kinds of mutation-rate settings to meet the requirements of different applications. Because of its features, the proposed processor is very suitable for various real-time applications. Finally, the proposed VLSI architecture is implemented on FPGA for verification.

Keywords: Genetic Algorithm(GA), Flexible design, Table-Look-Up Technique.

1. Introduction

The genetic algorithm (GA) can find an optimal solution by natural selection, so it has been used widely in many complex applications such as image processing, fuzzy control, neural network, communication system, and layout optimization [1-3]. Generally, genetic algorithm requires very intensive computations in order to perform the optimization. Hence, the study of specified VLSI implementation for it is very important and inevitable. In the past few years, many VLSI architectures for genetic algorithm have been proposed [4-7].

Most previous architectures of GA intended to improve the processing speed or to reduce the hardware cost. The main drawback of those architectures is that each of them can implement only one specified fitness function in its architecture. Actually, different applications require different GA fitness functions. To solve the problem, we propose a flexible genetic algorithm processor in this paper. It can perform dynamically various fitness functions, four crossover operations, and over ten thousand kinds of mutation-rate settings to meet the requirements of different applications. Because of its features, the proposed processor is very suitable for various real-time applications.

The paper is organized as follow. In Section 2, some basic concepts of GA are summarized. Section 3 describes the proposed VLSI architecture in detail. In Section 4, the comparisons of different GA processors are presented. Conclusions are provided in Section 5.

2. Genetic Algorithm

Figure 1 shows the flowchart of GA. It consists of six main steps: population initialization, fitness calculation, termination judgment, selection, crossover, and mutation. At the beginning, the initial population for GA is generated randomly. Then the evaluation values of fitness function of each individual in current population are calculated. After that, the termination criterion is checked. If the termination criterion is reached, the whole GA procedure stops; otherwise, the following three steps will be performed.

The selection works as the nature's survival of the fitness process. The fitness values of all individuals are evaluated and the elite are selected. In other words, fitter solutions survive while weaker ones perish in this step. After selection, crossover and mutation operations are performed on the elite to generate the new individuals, treated as the next generation population. With the help of crossover and mutation operations, we can avoid converging to the local optimum and locate the better solutions. Finally, the termination criterion is rechecked to decide whether the GA procedure should continue.

3. The proposed VLSI architecture

The GA VLSI architectures, proposed in [4] and [5], use redesign method to implement different fitness functions. In [6] and [7], they generate different fitness functions by using re-programmable FPGA board. Since different optimization problems require different fitness functions, those previous architectures are not very suitable. In order to increase the flexibility of implementing various fitness functions, we propose a GA processor based on the table-look-up technique. Besides, our processor can perform four crossover operations and over ten thousand kinds of mutation-rate settings to meet the requirements of different applications. The block diagram for the proposed GA processor is shown in Fig. 2. Five main blocks are described in detail in the following sections.

3.1 Random Number Generator (RNG)

Generally speaking, there are two methods to implement the random number generator (RNG): linear feedback shift register (LFSR) or linear cellular automata (LCA). Most RNG are realized with LCA because it has been demonstrated to generate better random sequences than LFSR [8]. Hence, we adopt LCA method to generate necessary individuals. Besides, two most popular rules, Rule-90 and Rule-150, in LCA are used to realize the RNG. The rule-90 operation is given as $s_i^+ = s_{i-1} \oplus s_{i+1}$, where s_i^+ denotes the next state for site s_i. The rule-150 operation is given as $s_i^+ = s_{i-1} \oplus s_i \oplus s_{i+1}$. The two rules can be implemented with the circuits shown in Fig. 3(a) and 3(b) respectively. Figure 4 shows the block diagram of 14-bits RNG architecture.

3.2 Selection Module (SM)

Selection module (SM) is one of the most important operations in genetic algorithm (GA). Here, we adopt tournament selection for this operation, and illustrate its architecture in Fig. 5. It consists of two parts: initial selection module (ISM) and selection record module (SRM). The ISM, the upper block of Fig. 5, will execute tournament selection and compare the fitness values of two individuals every time. According to different applications, users can decide to select the maximum or minimum by using sel_com. After tournament selection, the two advantageous (or survival) individuals are stored in the winner individual A' and B' registers and sent to the crossover module. At the same time, Mux2 can be used to select a proper control signal to the following SRM to determine whether replacing the individuals in the population memory. Finally, the best individuals will be decided and stored in a register named as Best Individual Register (BIR). The SRM, the bottom block of Fig. 5, will be used to record the addresses of worse individuals and execute the replacement operations.

3.3 Crossover Module (CM)

Crossover module (CM) is used to perform the crossover operation on two winner individuals **A** and **B**. Fig. 6 show that the block diagram of crossover module respectively. In the design, we offer four crossover operations including uniform crossover, single point crossover, two points crossover and cross crossover. Users can choose one of them according to their needs. The output chromosomes denoted as **A'** and **B'** are send to the following mutation module.

3.4 Mutation Module (MM)

Mutation module (MM) is quite important in GA. Figure 7 shows its hardware structure. It consists of two registers, fourteen comparators. The mutation operation is used to avoid converging to the local optimum and locate the better solutions [9]. Here, a flexible mutation-rate settings scheme is used. The ranges of dynamic mutation rate are from 1/16383 to 1. Users can choose an appropriate mutation rate dynamically and easily based on their needs. In the design, the mutation operation is performed when the user-defined mutation rate exceeds the threshold (generated by RNG and stored in the 14-bit shifter register), and is used to generate the new chromosome. Finally, the new chromosomes are generated and feed into the population memory.

3.5 Fitness Module (FM)

In order to perform the calculations of various fitness functions quickly and efficiently, our processor adopts the table-look-up technique. Different fitness functions can be implemented with the pre-designed software programs, and then the corresponding output values can be calculated and stored into the tables. Those various and complex fitness functions can be realized easily in our processor with the manner of table mapping, and the computation time as well as the complexity required for fitness calculation can be reduced largely.

4. Comparisons and Implementation

Table 1 shows the comparisons of different GA processors. Our processor is the only one that adopts dynamic mutation-rate settings. With the table-look-up technique, the proposed processor can perform various fitness functions easily and quickly. Finally, the proposed GA processor is realized with Verilog hardware description language. Figure 8 shows the layout of the processor. To further verify our design, we implemented the processor on FPGA and integrated it to a completed demo system shown in Fig. 9. Simulation results show that it works very well.

5. Conclusions

Genetic algorithm requires very intensive computations in order to perform the optimization. Hence, a dedicated VLSI implementation for it is necessary. In this paper, we propose a more flexible genetic algorithm processor architecture that can perform various fitness functions both quickly and dynamically to meet the requirements of different applications. Because of the features, the proposed processor is very suitable for various real-time applications.

References

[1] J.-M. Rouet, J.-J. Jacq, and C. Roux, "Genetic algorithms for a robust 3-D MR-CT registration," *IEEE Trans. on Information Technology in Biomedicine*, vol. 4, pp. 126-136, June 2000.

[2] C.-C. Chen, and C.-C. Wong, "Self-generating rule-mapping fuzzy controller design
using a genetic algorithm," *Proc. of IEE*, vol. 149, pp. 143-148, March 2002.

[3] C. Ergün, and K. Hacioglh, "Multiuser Dete- ction using a genetic algorithm in CDMA communications systems," *IEEE Trans. on Communications*, vol. 48, no. 8, pp. 1374-1383, Aug. 2000.

[4] S. D. Scott, A. Samal, and S. Seth, "HGA: A hardware based genetic algorithm," *ACM/SIMDA 3rd Int. Symposium on FPGA*, pp. 53-59, 1995.

[5] N. Yoshida, and T. Yasuoka, "Multi-GAP: Parallel and distributed genetic algorithm in VLSI," *IEEE Int. Conf. on System, Man, and Cybernetics*, vol. 5, pp. 571-576, 1999.

[6] S. Wakabayashi, T. Koide, K. Hatta, Y. Nakayama, M. Goto, and N. Toshine, "GAA: a VLSI genetic algorithm accelerator with on-the-fly adaptation of crossover operators," *IEEE Int. Symposium on Circuits and Systems*, vol. 2, pp. 268-271, 1998.

[7] J. J. Kim, and D. J. Chung, "Implementation of genetic algorithm based on hardware optimization," *IEEE Region 10 Conf. (TENCON)*, vol. 2, pp.1490-1493, 1999.

[8] M. Serra, T. Slater, J. C. Muzio, and D. M. Miller, "The analysis of one-dimensional linear cellular automata and their aliasing properties," *IEEE Tran. on CAD*, vol. 9, pp. 767-778, July 1990.

[9] R. L. Haupt, "Optimum population size and mutation rate for a simple real genetic algorithm that optimizes array factors," *in Proc. IEEE Int. Symposium*, vol. 2, pp. 1034-1037, 2000.

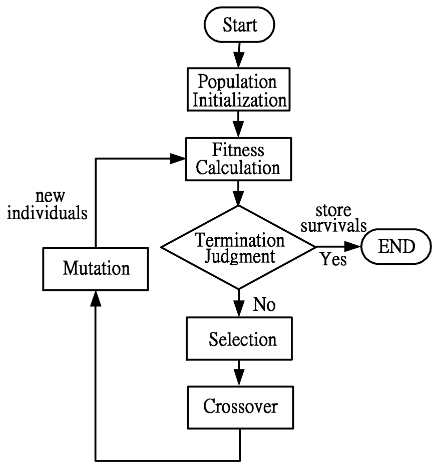


Fig. 1. The flowchart for GA.

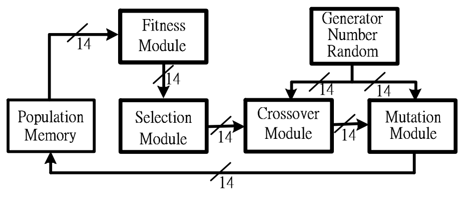


Fig. 2. Block diagram for the proposed GA processor.

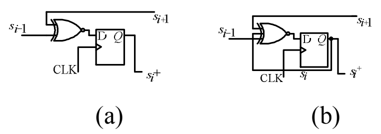


Fig. 3. a) Rule-90, and b) Rule-150.

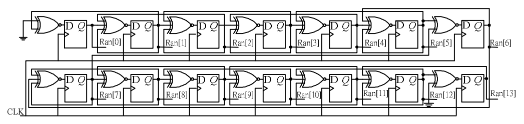


Fig. 4. 14 bits RNG based on cellular automata.

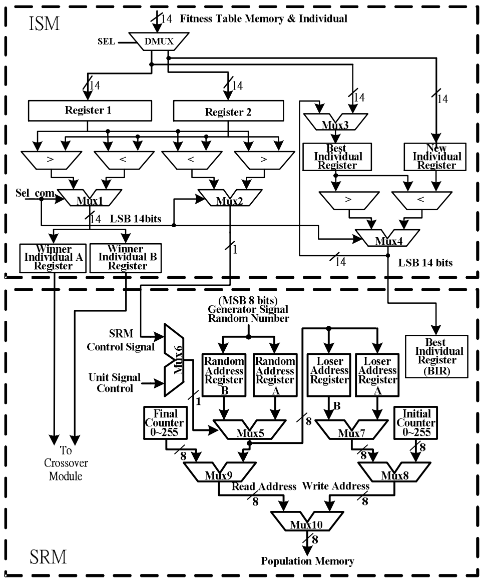


Fig. 5. Hardware structure of selection module.

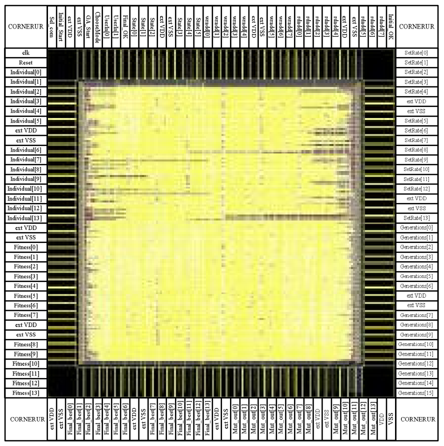


Fig. 8. The layout of the processor.

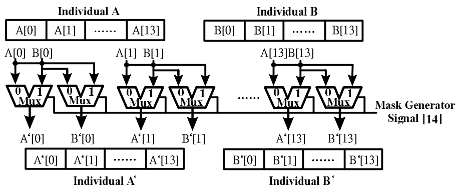


Fig. 6. Hardware structure of crossover module.

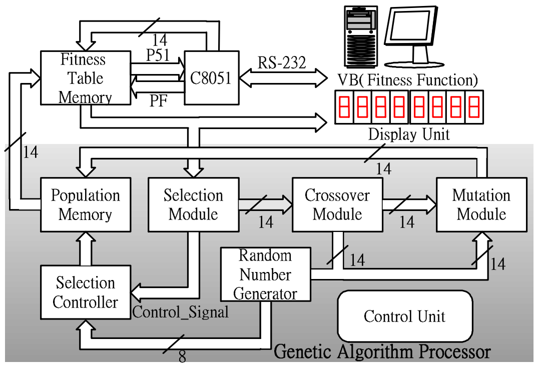
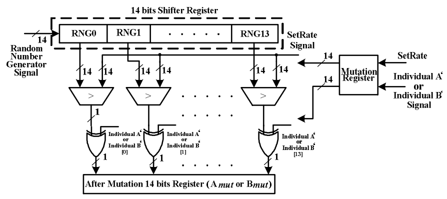


Fig. 9. The demo system.

Table 1. Comparison of various GAPs.

	Selection	RNG	Fitness Module	Mutation Rate
[4]	Roulette	CA	Redesign	None
[5]	Simplified Tournament	CA	Re-Program	None
[6]	Roulette & Elitist strategy	CA	Re-Program	Without Architecture
[7]	Tournament	CA	Re-Program	Single Point Multipoint
Our	Tournament	CA	Look Up Table	Dynamic


14 bits Shifter Register
RNG0
RNG1
RNG13
SetRate Signal
Random Number Generator Signal
SetRate
Mutation Register
Individual A' or Individual B' Signal
After Mutation 14 bits Register (Amut or Bmut)

F

ig. 7. Hardware structure of mutation block.