

Design Electronic Circuits Using Evolutionary Algorithms

Xuesong Yan¹, Qinghua Wu², Chenyu Hu¹ and Qingzhong Liang¹

¹*School of Computer Science, China University of Geosciences, Wuhan, China*

²*School of Computer Science and Engineering, Wuhan Institute of Technology, Wuhan, China*
yanxs@cug.edu.cn

doi: 10.4156/jnit.vol1.issue1.11

Abstract

For the evolutionary algorithm, the representation of the electronic circuit has two methods, one kind is code with the electronic circuit solution space, the other is code with the problem space. How one representation quality weighs may think the following questions? The first is the code method should as far as possible complete, it is say for the significance solution circuit or the optimize solution obtains in the problem space may represented by this code method. The second is the code method should speeds up the convergence speed of the algorithm search. The hardware representation methods mainly include binary bit string representation, tree representation, Cartesian Genetic Programming representation and other representations. In this paper, we introduce the representations of the binary bit string and Cartesian Genetic Programming in detail, and based on the two representations we design the evolutionary algorithm for electronic circuits optimization. Then we introduce the encoding of the circuit as a chromosome, the genetic operators and the fitness function of our algorithm. For the case studies showed this algorithm has proved to be efficient, and the experiment results showed that we have gained better results.

Keywords: *Electronic circuits, Evolutionary algorithms, Evolvable hardware*

1. Introduction

Evolutionary Electronics applies the concepts of genetic algorithms to the evolution of electronic circuits. The main idea behind this research field is that each possible electronic circuit can be represented as an individual or a chromosome of an evolutionary process, which performs standard genetic operations over the circuits. Due to the broad scope of the area, researchers have been focusing on different problems, such as placement, Field Programmable Gate Array (FPGA) mapping, optimization of combinational and sequential digital circuits, synthesis of digital circuits, synthesis of passive and active analog circuits, synthesis of operational amplifiers, and transistor size optimization. Of great relevance are the works focusing on “intrinsic” hardware evolution in which fitness evaluation is performed in silicon, allowing a higher degree of exploration of the physical properties of the medium. This particular area is frequently called Evolvable Hardware[1-4].

In the sequence of this work, Coello, Christiansen and Aguirre presented a computer program that automatically generates high-quality circuit designs [5]. They use five possible types of gates (AND, NOT, OR, XOR and WIRE) with the objective of finding a functional design that minimizes the use of gates other than WIRE (essentially a logical no-operation).

Miller, Thompson and Fogarty applied evolutionary algorithms for the design of arithmetic circuits. The technique was based on evolving the functionality and connectivity of a rectangular array of logic cells, with a model of the resources reflecting the Xilinx 6216 FPGA device [6]. Kalganova, Miller and Lipnitskaya proposed another technique for designing multiple-valued circuits. The EH is easily adapted to the distinct types of multiple-valued gates, associated with operations corresponding to different types of algebra, and can include other logical expressions [7]. In order to solve complex systems, Torresen proposed the method of increased complexity evolution. The idea is to evolve a system gradually as a kind of divide-and-conquer method. Evolution is first undertaken individually on a large number of simple cells. The evolved functions are the basic blocks adopted in further evolution or assembly of a larger and more complex system [8].

More recently Hollingworth, Smith and Tyrrell describe the first attempts to evolve circuits using the Virtex Family of devices. They implemented a simple 2-bit adder, where the inputs to the circuit

are the two 2-bit numbers and the expected output is the sum of the two input values [9]. Based on the Miller's method, Yan applied Gene expression programming (GEP) for the design of electronic circuits and the case study shows this technology was effective[10-11].

A major bottleneck in the evolutionary design of electronic circuits is the problem of scale. This refers to the very fast growth of the number of gates, used in the target circuit, as the number of inputs of the evolved logic function increases. This results in a huge search space that is difficult to explore even with evolutionary techniques. Another related obstacle is the time required to calculate the fitness value of a circuit [12]. A possible method to solve this problem is to use building blocks either than simple gates. Nevertheless, this technique leads to another difficulty, which is how to define building blocks that are suitable for evolution.

Now the research of the evolvable hardware lies in the hardware representation and the concrete example, not the algorithm, so the hardware representation and code is the most important research contents. No matter is carries on the electronic circuit design and optimize using the evolvable hardware, or carries on the evolvable hardware research, the basic problems are the domain knowledge and hardware representation. For the evolutionary algorithm, the representation of the electronic circuit has two methods, one kind is code with the electronic circuit solution space, the other is code with the problem space. How one representation quality weighs may think the following questions? The first is the code method should as far as possible complete, it is say for the significance solution circuit or the optimize solution obtains in the problem space may represented by this code method. The second is the code method should speeds up the convergence speed of the algorithm search. The hardware representation methods mainly include binary bit string representation, tree representation, Cartesian Genetic Programming representation and other representations.

Following the line of research, this paper using evolutionary algorithms for designing electronic circuits. This paper is organized as follows. Section 2 introduces the evolutionary algorithms (EA). Section 3 introduces the algorithms to design the circuit based on binary bit string representation. Section 4 introduces the algorithms to design the circuit based on Cartesian Genetic Programming representation. Finally, section 5 presents the main conclusions.

2. Evolutionary Algorithms

Evolution can be viewed as a change in the genetic composition of a population of individuals over time. In a simplified form, evolution is result of the successive processes of reproduction and genetic variation followed by natural selection, which allows the fittest individuals to survive and reproduce, thus propagating their genetic material to future generations.

By mimicking the process of natural evolution, researchers developed the evolutionary algorithms (EA), which are based on the collective adaptability within a population of individuals, each of which represents a search point in the space of potential solutions to a given problem. In order to an evolutionary algorithms to work, a population of candidate solution is initialized, and it evolves towards increasingly better regions of the search space by means of selection, reproduction and genetic variation mechanisms. The environment in which the population evolves is defined by the aim of the search, and delivers an information, termed fitness, that quantifies how good is an individual. The selection process favors the reproduction of individuals of higher fitness, and a recombination mechanism allows the mixing of parental information while passing it to their descendants. Finally, mutation introduces novelties in the population.

There are three main types of evolutionary algorithms: 1) evolutionary strategies; 2) genetic algorithms, and 3) evolutionary programming [13]. Note that some authors consider genetic programming and classifier systems as other branches of the evolutionary algorithms; a discussion that will not be pursued here.

From a practical perspective, evolutionary algorithms are aimed at performing a search to identify an approximation of an (global) optimum of an objective function. The search is performed by evolving a population of individuals represented, in most cases, according to the application domain and type of evolutionary algorithms. The individuals of the population correspond to chromosomes that during the evolutionary process are allowed to suffer genetic variation and/or exchange genetic material.

Evolutionary algorithms are a kind of ways, which simulate the evolutionary processes of the nature, especially the evolutionary processes of creatures by computers for solving complex problems. The EA has such characteristics as self-adaptation, self-organization, self-learning, self-optimization and intrinsic parallelism. Using it can get rid of the needs of knowledge bases and formal logic deduction in some sense and may realize automatic programming.

2.1. Intelligent

The intelligent of evolutionary algorithms includes: self-organization, self-adaptation, self-learning and so on. Use EA solve problem, if the code scheme, fitness function and genetic operator have confirmed, the algorithm may be self-search with the information that achieved from the process of evolution. Because the natural selection strategy, which allows the fittest individuals to survive and reproduce, then the fittest individuals have the high survive probability. In common, the fittest individuals have more adaptation to environment with their gene structure, then through the crossover operator and mutation operator, propagating their offspring, fitting into the environment very much. The EA has such characteristics as self-adaptation, self-organization and so on. Using it can discovery the environment character and rule automatic according the environment changes.

2.2. Modeling

Evolutionary algorithms has such characteristics as self-learning and self-organization, it can help us automatic find the rules from data and build models. For example, advances in the areas of trend analysis, periodicity analysis, sequential pattern analysis, or similarity search in time series [8] could be of immediate benefit to data mining.

2.3. Parallelism

The evolutionary algorithms' intrinsic parallelism includes two aspects: the first is inherent parallelism, EA itself adapt to parallelism. The simplest fashion is each of the hundreds or thousands computer process independent population's evolutionary computation; in the processing each populations have no communication (communication would attain better individual). At the end of the computation, select the best individuals with communication. This parallelism fashion has no limit for the structure of parallelism system. It is to say, EA adapt to process on the distributed systems at present, and have no effect to the parallelism efficiency. The second is implicit parallelism, EA use population organize search, so it can search many areas of the result space simultaneity and change the information with each other. This search fashion cause evolutionary computation attains more benefit with little compute.

3. Evolutionary Algorithms Based on Binary Bit String Representation

3.1. Binary Bit String Representation

Programming Logic Device (PLDs) are hardware devices of which architecture can be determined by downloading a binary bit string, called architecture bits. Architecture bits are the compilation result from higher level of hardware description such as boolean functions and truth tables. On the other hand, genetic algorithms (GA) are robust search algorithms where multiple chromosomes (usually represented as a binary bit string) are used to find better solutions.

The basic idea of EHW [14] is to regard the architecture bits of PLDs as chromosomes of GAs and to find out better hardware structure by GA, as shown in Figure 1. Once a good chromosome is found by GA, it is downloaded into PLDs on the spot. A PLD consists of logic cells and a fuse array. A logic cell can realize some logic functions and one of the functions is selected by specifying a bit string. In other words, the logic cell's function is programmable.

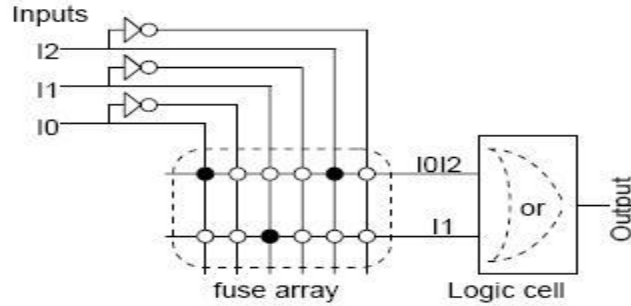


Figure 1. A simplified PLD(Programmable Logic Device) structure

Fuse array determines the interconnection between external inputs and logic cells. It actually generates AND-term signals by specifying the switches on the fuse array. For example, in Figure 2, the black dots on the fuse array indicate that external inputs are connected with a line of the fuse array. The switch setting on the first row of Figure 2 indicates that I0 and I2 are entered into the first row, generating a AND-term, I0, I2. Similarly, I1 is generated on the second row. These AND-terms then enters into the logic cell of which function is chosen to be an OR gate. Notice that such switch setting are also regarded as a binary bit string. Thus, both of the fuse array and logic cell functions are controlled by a binary string. The key idea of EHW is to regard such a binary bit string as chromosome of GA and find out the best hardware structure by GA.

There are some research work on EHW was directly focused on the architecture bits[9]; bits for fuse array and bits for logic cells are regarded as genotype. Although this genotype representation is straightforward, the researchers succeeded in hardware evolution of both combinatorial logic circuits and sequential logic. However, this genotype representation has inherent limitations, since the fuse array bits are fully included in the genotype even in the case that only a few bits are effective. This causes the increase of the chromosome length, increasing the GA execution time.

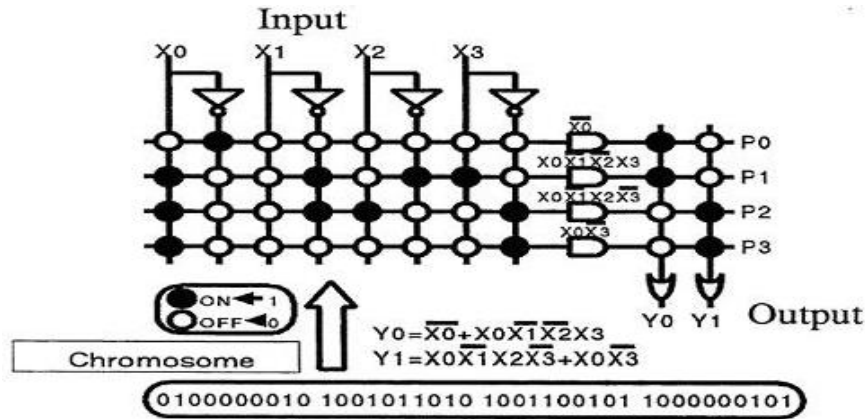


Figure 2. Chromosome of PLD

3.2. Modeling for Circuit representation

For this representation we design the algorithm, in our algorithm, the AND-gate array with M column and K row, the OR-gate array with N column and K row, then in the model have two array with $M \times K$ and $N \times K$, the length of the chromosome will be calculate by this equation: $\text{Length} = (2 \times M + N) \times K$.

In here, we give a example, for a circuit with four input and one output. With our model, the length of the chromosome is 36, the initial bit-string with constraints: in the bit-string the two adjoin bits will not be one in the same time. For the following individual, the red denotes the OR-gate.

individual: 01001010110101000000100101010010001101

For this individual we can know the logic function is: $F = ACD + \overline{A}\overline{C}\overline{D}$ and the circuit struction is like Figure 3.

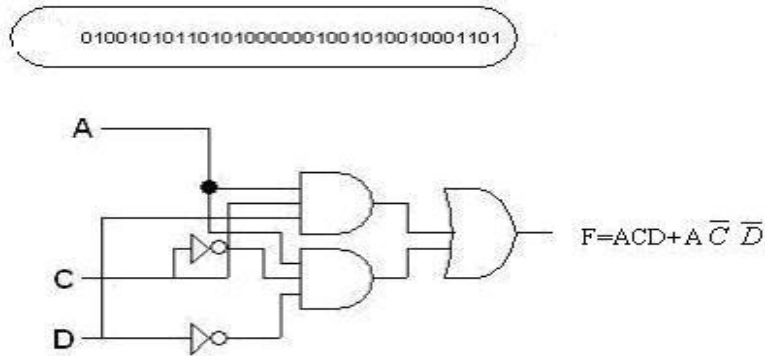


Figure 3. Circuit and Logic Expression

3.3. Individual Evaluate and Operator Design

In the algorithm, the most important is the fitness function, because only the fitness function can define the individual good or bad, so in our algorithm, we describe the fitness function like this: we can get the logic function from the bit-string and detect the function maybe fit the truth table, then calculate the gates number of the corresponding logic circuit, the gates number more little and the fitness value more best. The whole process is like this: According the chromosome bit-string, we can get a logic function expression, we can put the values in the truth table into the expression and calculate a value. If the value is equal to the output then we know the individual is the right individual. If the input variant have four: A、B、C、D, then we can statistic the number of using the four variants A、B、C、D and the number of using the logic gates. In the evolution process, we must maintain the individual, which has fewer variants, logic gates and can Satisfies the function.

In our algorithm, we deign the genetic operator according the circuit function. The crossover method we use the two-position crossover and the mutation we use one point mutation. The length of the chromosome is 36 bit, the process of crossover operation like following:

Individual 1 : 100001010101010011100010101010000001

Individual 2: 001010000010001101011001001100110000

In the crossover, we select two number: m and n in random, in here $m < 36$, $n < 36$.

If $m < n$, we exchange the string from n to m in individual 1 and individual 2, then generate the two new individuals.

New individual 1: 100001000010001101100010101010000001

New individual 2: 001010010101010011011001001100110000

The mutation operation like the crossover, the step is following:

individual: 100001010101010011100010101010000001

generate the number m in random, in here $m < 36$.

For this operation, we will find the value a according the point m and do the operation 1-a, the new individual is like this:

New individual: 1000010100101010011100010101010000001

3.4. Algorithm Framework

```
begin
{
```

```

Generate the population of the individual PopSize P(0), t=0;
Calculate the fitness values of the individuals;
while(< Max-generation) Do
{
    do the genetic operation with the individual;
    according the fitness value and selection strategy calculate the selection probability Pi and the
crossover probability Pc of the individuals;
    do the crossover operation according to the Pi and calculate the child individual fitness value, if the
fitness value is better then the child population replace the parent individual and update the population;
    do the mutation operation according to the Pi to select N1(N1<Popsi)e)individual, and calculate the
fitness value, if the fitness value is better then the child population replace the parent individual and
update the population;
    generate the new population P(t+1);
    t=t+1 ;
}
}
    
```

3.5. Experiment result

For the model and the operator, we test our algorithm use the following example. In here the PLD model is four inputs and one output, and the test function is ture-table, the input is: A, B, C, D and the output is F.

Table 1. Ture-table of the problem

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

According to the ture-table, we use our algorithm design the circuit, the parameters for the algorithm are as Table 2.

Table 2. The parameters for the algorithm

Number of generations	1000
Population Size	100
Mutation rate	0.003
Crossover rate	0.08
Selection range	100

In the Table 3 are the experiment results.

Table 3. Experiment results

Individual string	Number of the result	Number of Gate	Number of Variant
100010000101000001101010001100010001	16	4	3
100010101101000000100010001101000000	16	4	4
101001001100010001101010000101010000	16	5	4
101001101100010001100010001101000001	16	6	5
010100100100010001101000101101010100	15	21	4
101000011100010101001010100100100110011	15	21	6
10000000110101010110101010110101010101	14	21	4
100000001101010101101010101100101011	14	21	7

From the experiment results, we can know use the algorithm design circuit can get different design scheme, and can verify our algorithm is effective for the problem of circuit design.

4. Evolutionary Algorithms Based on Cartesian Genetic Programming Representation

4.1. Cartesian Genetic Programming Representation

In our algorithm, the chromosome representation we use Miller's[15]. This representation is based on the FPGA of Xilinx Virtex-II. The starting point in this technique is to consider, for each potential design, a geometry (of a fixed size array) of uncommitted logic cells that exist between a set of desired inputs and outputs. The uncommitted logic cells are typical of the resource provided on the Xilinx FPGA part under consideration. An uncommitted logic cell refers to a two-input, single-output logic module with no fixed functionality. The functionality may then be chosen, at the implementation time, to be any two input variable logic function. In this technique, a chromosome is defined as a set of interconnections and gate level functionality for these cells from outputs back toward the inputs based upon a numbered rectangular grid of the cells themselves, as in Figure 4 . The inputs that are made available are logic '0', logic '1', all primary inputs and primary inputs inverted. To illustrate this consider a 3 x 3 array of logic cells between two required primary inputs and two required outputs.

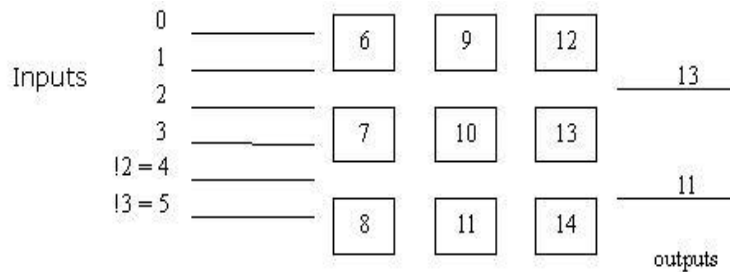


Figure 4. A 3 x 3 geometry of uncommitted logic cells with inputs, outputs and netlist numbering

The inputs 0 and 1 are standard within the chromosome, and represent the fixed values, logic '0' and logic '1' respectively. The inputs (two in this case) are numbered 2 and 3, with 2 being the most significant. The lines 4 and 5 represent the inverted inputs 2 and 3 respectively. The logic cells which form the array are numbered column-wise from 6 to 14. The outputs are numbered 13 and 11, meaning that the most significant output is connected to the output of cell 13 and the least significant output is connected to the output of cell 11. These integer values, whilst denoting the physical location of each input, cell or output within the structure, now also represent connections or routes between the various points. In other words, this numbering system may be used to define a netlist for any combinational circuit. Thus, a chromosome is merely a list of these integer values, where the position on the list tells us the cell or output which is being referred to, while the value tells us the connection (of cell or input) to which that point is connected, and the cells functionality.

Each of the logic cells is capable of assuming the functionality of any two-input logic gate, or, alternatively a 2-1 multiplexer (MUX) with single control input. In the geometry shown in Figure 5. a sample chromosome is shown below:

0 2 -1 1 3 -5 2 4 3 0 8 -10 7 8 -4 6 11 9 6 4 -9 2 11 7 13 11

Figure 5. A typical netlist chromosome for the 3 x 3 geometry of Figure 1

Notice, in this arrangement that the chromosome is split up into groups of three integers. The first two values relate to the connections of the two inputs to the gate or MUX. The third value may either be positive - in which case it is taken to represent the control input of a MUX - or negative - in which case it is taken to represent a two-input gate, where the modulus of the number indicates the function according to Table 4 below. The first input to the cell is called A and the second input called B for convenience. For the logic operations, the C language symbols are used: (i) & for AND, (ii) | for OR, (iii) ^ for exclusive-OR, and (iv) ! for NOT. There are only 12 entries on this table out of a possible 16 as 4 of the combinations: (i) all zeroes, (ii) all ones, (iii) input A passed straight through, and (iv) input B passed straight through are considered trivial - because these are already included among the possible input combinations, and they do not affect subsequent network connections in cascade.

Table 4. Cell gate functionality according to negative gene value in chromosome

Gene Value	Gate Function
-1	A & B
-2	A & !B
-3	!A & B
-4	A ^ B
-5	A B
-6	!A & !B
-7	!A ^ B
-8	!A
-9	A !B
-10	!B
-11	!A B
-12	!A !B

This means that the first cell with output number 6 and characterised by the triple {0, 2, -1} has it's A input connected to '0', its B input connected to input 2, and since the third value is -1, the cell is an AND gate (thus in this case always produces a logical output of 0). Picking out the cell who's output is labelled 9, which is characterised by the triple {2, 6, 7}, it can be seen that its A input is connected to input 2 and its B input is connected to the output of cell 6, while since the third number is positive the cell is a MUX with control input connected to the output of cell 7.

4.2. Genetic Operation

Mutation Operation: In this algorithm, the mutation operate is for the triple, each or some of the gene in the triple do the mutate, the mutation probability is 0.7, that is to say, each triple do the mutate with this probability and the mutation domain is 20.

Crossover Operation: Select the fittest individual being the parent for the next generation in the population. Each other individual do the crossover operate with this parent.

Fitness Measure: The test of whether the evolved circuits perform the desired logic translation of inputs to outputs is achieved by running all test inputs through the network, and compared the results with the desired functionality in a bit-wise fashion. A PLA file (truth table) contains the target function, and this is used as a basis for comparison. The percentage of total correct outputs in response to appropriate inputs is then used as the fitness measure for the genetic algorithm. In other words, the nearer the evolved circuit comes to performing with desired functionality, the fitter it is deemed to be.

In our algorithm we use this method: comparing all the outputs of an individual with the desired outputs. For a set of input in the true table, if there is one bit which belongs to the outputs of the circuit individual, different to the desired value, though deemed the circuit functionality useless for this input. When total outputs are equal to the desired outputs, then the fitness value added 1.

4.3. Algorithm Framework

Our algorithm through populations evolution to get the fittest solution, in the evolutionary process, we use the genetic operation to guarantee the individual diversity in the population, thus enables the population to be able rapid convergence. The electric circuit automation designs problem has it's particularity, compared with other optimized problem the different place for the electric circuit automation design lies in the ultimate objective is explicit which is to get the circuit solution, in this circuit the output of a individual fit the true table value. For this idea, we design our algorithm like following:

Step 1: random generation of the initial populations P10 and initial the counter $t=0$;

Step 2: select the individuals from populations according to the fitness value and crossover with these individuals to get the new individuals Q;

Step 3: for the new individuals use the mutation operator according to probability to generate the new population P_t;

Step 4: $t=t+1$;

Step 5: if the terminate conditions satisfied turn to step 6, else turn to step 2;

Step 6: output the new population P_t;

From the framework with our algorithm, it is show this algorithm has guaranteed the population's diversity and causes the population not to trap into the local optima.

4.4. Experiment result

This section shows the implementation of our experiment case: one-bit full adder circuit, tow-bit full adder circuit, three-bit full adder circuit and four-bit full adder circuit. The parameters for the algorithm are as Table 5.

Table 5. The parameters for the algorithm

Number of generations	2000
Population Size	30
Chromosome length	39
Mutation rate	0.7

4.4.1. One-bit full adder

Evolving the one-bit adder was easier to do on a larger geometry but resulted in a less efficient circuit. That is many genetic algorithm was able to discover 100% functional solutions was intimately related to the size of the geometry, but our algorithm use small geometry to find the fully functional solutions.

The Figure 6 is the one-bit adder without using optimum algorithm, and our resulting circuits as shown in Figure 7. From the figure we know it is a gratifying result to obtain as it is clear that this design is an optimum solution.

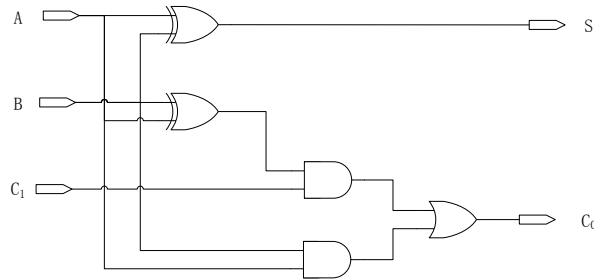


Figure 6. One-bit full adder without optimum

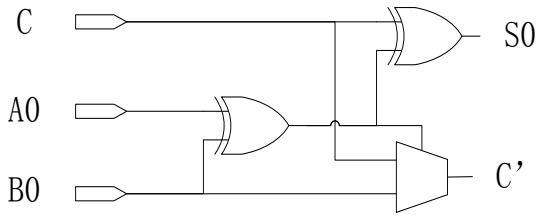


Figure 7. The evolved optimum one-bit full adder circuit

4.4.2. Two-bit full adder

A two-bit full adder, with a truth table with 5 inputs and 3 outputs. In this case, Our algorithm use small geometry to find the fully functional solutions, the matrix has a size of 3×3 . The original circuit is showed in Figure 8 and our resulting circuits as shown in Figure 9. From the figures we know it is a gratifying result to obtain as it is clear that this design is an optimum solution.

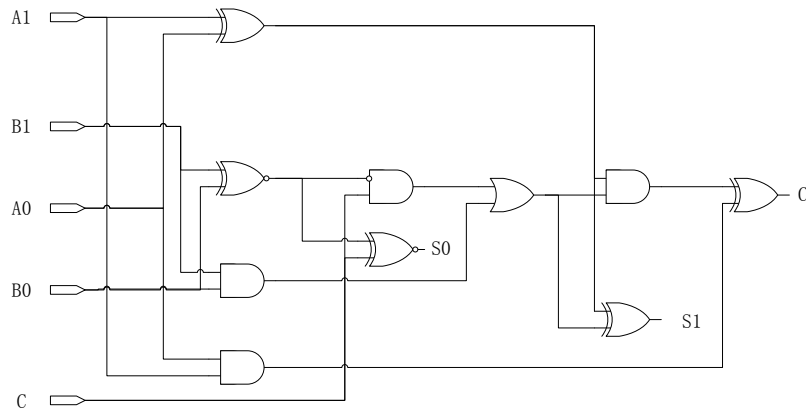


Figure 8. Two-bit full adder without optimum

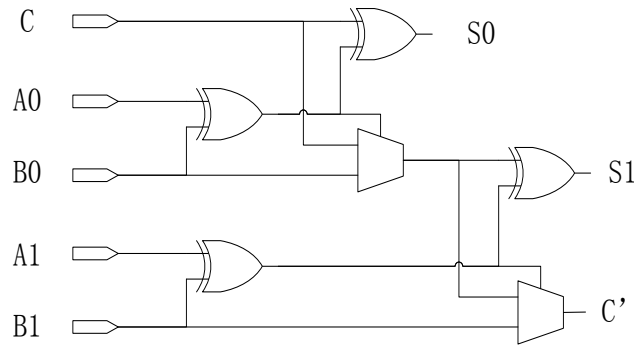


Figure 9. The evolved optimum two-bit full adder circuit

4.4.3. Three-bit full adder

A three-bit full adder, with a truth table with 7 inputs and 4 outputs. In this case, Our algorithm use small geometry to find the fully functional solutions, the matrix has a size of 4×4 . The original circuit is showed in Figure 10 and our resulting circuits as shown in Figure 11. From the figures we know it is a gratifying result to obtain as it is clear that this design is an optimum solution.

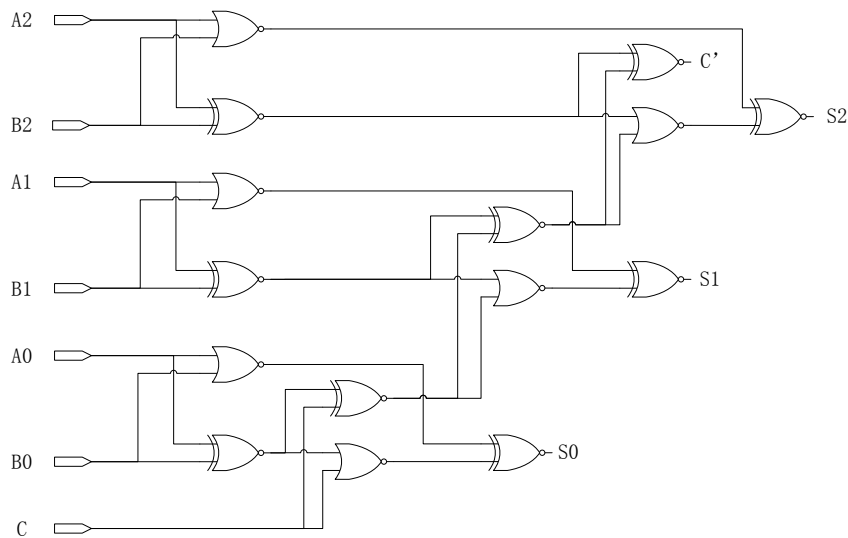


Figure 10. Three-bit full adder without optimum

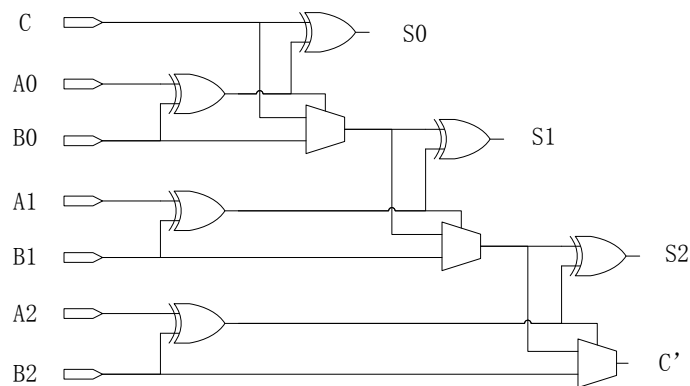


Figure 11. The evolved optimum three-bit full adder circuit

4.4.4. Four-bit full adder

A four-bit full adder, with a truth table with 9 inputs and 5 outputs. In this case, Our algorithm use small geometry to find the fully functional solutions, the matrix has a size of 8×8 . The original circuit is showed in Figure 12 and our resulting circuits as shown in Figure 13. From the figures we know it is a gratifying result to obtain as it is clear that this design is an optimum solution.

5. Conclusion

This paper proposed a new means for designing electronic circuits given a set of logic gates. The final circuit is optimized in terms of complexity (with the minimum number of gates). For the case studies this means has proved to be efficient, experiments show that we have better results.

There are still many avenues for further work. Other ways of representing rectangular arrays of logic cells may be devised and also, the relationship between cell connectivity and the evolvability of designs has still to be explored. There are many wider issues to be considered also which relate to the problem of evolving much larger circuits. It is a feature of the current technique that one has to specify the functionality of the target circuit using a complete truth table, however this is impractical for circuits with large numbers of inputs.

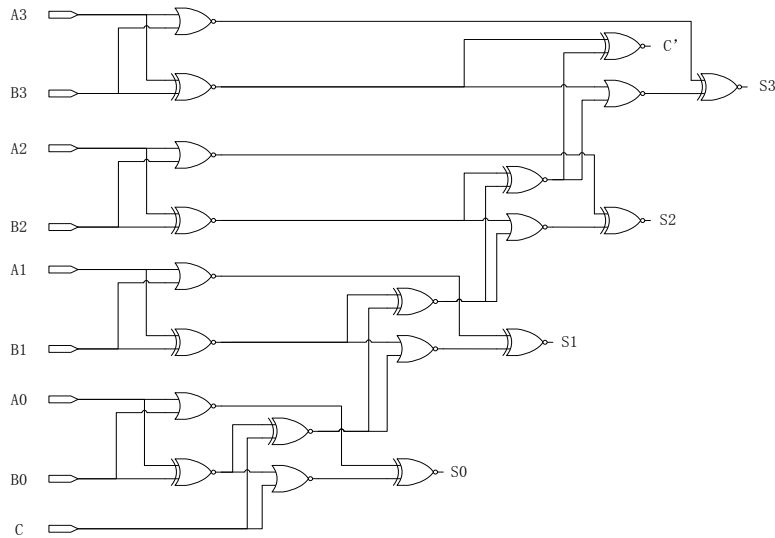


Figure 12. Four-bit full adder without optimum

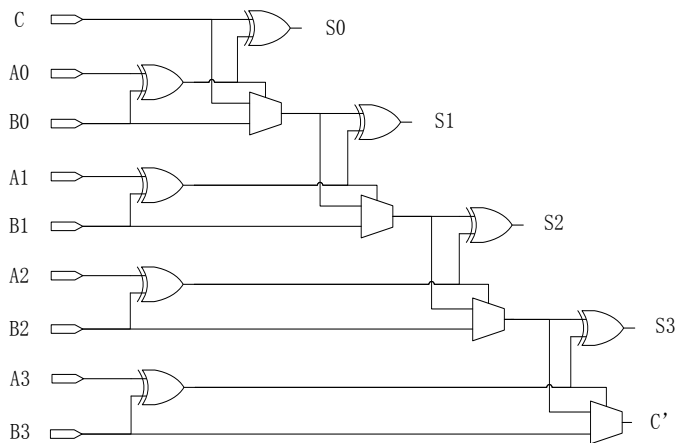


Figure 13. The evolved optimum four-bit full adder circuit

6. Acknowledgements

This paper is supported by Astronautics Research Foundation of China (NO.C5220060318) and Supported by the Special Fund for Basic Scientific Research of Central Colleges, China University of Geosciences(Wuhan).

7. References

- [1] Zebulum, R. S., Pacheco, M. A. and Vellasco, M. M., "Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms", CRC Press, 2001.
- [2] Thompson, A. and Layzell, P., "Analysis of unconventional evolved electronics", Communications of the ACM, Vol. 42, pp. 71-79, 1999.
- [3] Louis, S.J. and Rawlins, G. J., "Designer Genetic Algorithms: Genetic Algorithms in Structure Design", in Proceedings of the Fourth International Conference on Genetic Algorithms, 1991.
- [4] Candida Ferreira, "Gene Expression Programming: Anew adaptive Algorithm for solving Problems", Complex Systems. Vol. 13, 87-129, 2001.
- [5] Koza, J. R., "Genetic Programming: On the Programming of Computers by means of Natural Selection", MIT Press, 1992.
- [6] Coello, C. A., Christiansen, A. D. and Aguirre, A. H., "Using Genetic Algorithms to Design Combinational Logic Circuits", Intelligent Engineering through Artificial Neural Networks. Vol. 6, pp. 391-396, 1996.
- [7] Miller, J. F., Thompson, P. and Fogarty, T, Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications. Chapter 6, 1997, Wiley.
- [8] Kalganova, T., Miller, J. F. and Lipnitskaya, N., "Multiple-Valued Combinational Circuits Synthesised using Evolvable Hardware", in Proceedings of the 7th Workshop on Post-Binary Ultra Large Scale Integration Systems, 1998.
- [9] Torresen, J., "A Divide-and-Conquer Approach to Evolvable Hardware", in Proceedings of the Second International Conference on Evolvable Hardware. Vol. 1478, pp. 57-65, 1998.
- [10] X. S. Yan, Wei Wei et.al; "Design Electronic Circuits by Means of Gene Expression Programming" , Proceedings of the First NASA/ESA Conference on Adaptive Hardware and Systems, IEEE Press, pp. 194-199, 2006.
- [11] X. S. Yan et.al; "Designing Electronic Circuits by Means of Gene Expression Programming II", Proceedings of the 7th International Conference on Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science, Springer Press, pp. 319-330, 2007.
- [12] Vassilev, V. K. and Miller, J. F., "Scalability Problems of Digital Circuit Evolution" , in Proceedings of the Second NASA/DOD Workshop on Evolvable Hardware, pp. 55-64, 2000.
- [13] Z. Michalewicz, S. Esguvel et al., "The spirit of evolutionary algorithms", Journal of Computing and Information Technology, Vol.7, pp.1-18, 1999.
- [14] T. Higuchi, et al.: "Evolvable Hardware with Genetic Learning", Proceedings of Simulated Adaptive Behavior, The MIT Press, 1992
- [15] J. F. Miller, P. Thomson, "Cartesian Genetic Programming", Third European Conference on Genetic Programming Edinburgh, April 15-16, Proceedings published as Lecture Notes in Computer Science, Vol. 1802, pp.121-132, 2000.
- [16] X. S. Yan et.al; "Representations of Evolutionary Electronics", Proceedings of the 3rd International Symposium on Intelligence Computation & Applications, Lecture Notes in Computer Science, Springer Press, pp. 67-78, 2008.