Using Genetic Algorithms to Design Combinational Logic Circuits

Carlos A. Coello Coello Alan D. Christiansen Arturo Hernández Aguirre

Department of Computer Science
Tulane University
New Orleans, LA 70118

Abstract

We introduce a method, based on a Genetic Algorithm (GA) approach, to design combinational logic circuits. This problem is quite difficult for a traditional GA, but we have overcome these difficulties and have implemented a computer program that can automatically generate high-quality circuit designs. We describe the important issues to consider when solving this circuit design problem: the importance of the representation scheme, the encoding function, and the definition of the fitness function. We present several circuits derived by our system under various assumed constraints, such as the maximum number of allowable gates and the types of available gates. We compare the solutions produced by our system against those generated by a human designer. We also show that our representation approach, when compared to a standard binary encoding, produces better performance both in terms of quality of solution and in terms of speed of convergence.

1 Introduction

Design is usually considered to be an activity requiring considerable human creativity and knowledge. Even the definition of the term *design* itself is quite elusive, since it can be interpreted in several different ways depending on the task to be performed. Although there have been many attempts at developing programs for automated design, such programs are notoriously difficult to build.

In the research reported in this paper, we seek a computer-based tool that can make the design process less tedious for the human designer without sacrificing quality of the design produced. In this paper, we limit our focus to combinational logic circuits, which contain no memory elements. Such circuits contain no feedback paths.

2 Previous Work

A general search technique inspired by natural evolution, called the *genetic algorithm* [Holland, 1992], has been widely used for optimization tasks [Goldberg, 1989] and is known to be a very powerful tool in certain domains. In our current work we wish to find a way to use the genetic algorithm (GA) as a design tool, with particular emphasis in the design of combinational circuits.

Louis [Louis, 1993] is one of few sources found in the literature to address the use of GAs for the combinational logic design problem. In his dissertation [Louis, 1993] Louis combines knowledge-based systems with the genetic algorithm, making use of a genetic operator called *masked crossover* that adapts to the encoding, being able to exploit information unused by classical crossover operators. His results, although very encouraging for certain examples, do not seem to have solved the combinational circuit design problem completely. However, his idea of incorporating knowledge about the domain in the genetic operator constitutes a big step toward increasing the power of the GA as a design tool.

Koza [Koza, 1992] has used genetic programming to design combinational circuits. He has designed, for example, a two-bit adder, using a small set of gates (AND, OR, NOT), but his emphasis has been on generating functional circuits rather than on optimizing them. In fact, this is also the case in Louis' research, where the main focus was to provide an easier way to generate functional designs using the GA rather than in optimizing a functional design according to certain metrics. So far, genetic programming has been considered a more powerful tool in such tasks, because the representation it uses is more powerful for structural design in general.

In the work reported here, we are interested not only in producing functional designs, but also in optimizing them according to certain metrics. This is a quite complicated task for the GA, because we must deal with two difficult problems at the same time.

3 Statement of the Problem

The problem of interest to us consists of designing a circuit that performs a desired function (specified by a truth table), given a certain specified set of available logic gates. The complexity of a logic circuit is a function of the number of gates in the circuit. The complexity of a gate generally is a function of the number of inputs to it. Because a logic circuit is a realization (implementation) of a Boolean function in hardware, reducing the number of literals in the function should reduce the number of inputs to each gate and the number of gates in the circuit—thus reducing the complexity of the circuit. The algebraic method used to minimize functions is tedious and error prone. Its success depends on our ability to recognize the application of a theorem or a postulate during the minimization process. Such recognition may not be obvious. Furthermore, there is no general set of rules to aid that recognition.

Two popular minimization techniques are the *Karnaugh Map* [Karnaugh, 1953], which is based on a graphical representation of Boolean functions, and the *Quine-McCluskey Procedure* [Quine, 1955, McCluskey, 1956], which is a tabular method. Both of these methods are mechanical in nature. Karnaugh Maps are useful in minimizing functions with up to five or six variables. The Quine-McCluskey Procedure is useful for functions of any number of variables and can easily be programmed to run on a digital computer. Note that the algebraic simplification process depends entirely on one's familiarity with the postulates and theorems and one's ability to recognize their application. Of course, this ability varies from individual to individual. Depending on the sequence in which the theorems and postulates are applied, more than one simplified form of the expression may be obtained. Usually all such simplified forms are valid and acceptable. Thus, there is (in the general case) no single, unique minimized form of a boolean expression.

In this work, we compare the designs produced by a GA with those generated by a human designer using Karnaugh maps. The comparison is in many ways unfair because of differing capabilities of man and machine. For example, a human designer tends to use only the gates NOT, AND, OR and has more difficulties using XOR because the Karnaugh Map does not support the identification of XOR terms as well as it supports "seeing" simple product terms. The computer, using our GA approach, and not being restricted by human pattern recognition abilities, uses many XOR gates, often disregarding the NOT gate. Our overall measure of circuit optimality is the total number of gates used, regardless of their kind. This is approximately proportional to the total part cost of the circuit. Obviously, we perform this analysis for only fully functional circuits.

3.1 Example 1: A Two-bit Adder

We want to find the combination of five possible types of gates (AND, NOT, OR, XOR and WIRE) so that our circuit performs the two-bit addition of its inputs, producing a three-bit sum. Our objective is to find a functional design that minimizes the use of gates other than WIRE (essentially a logical no-operation).

The circuit can be represented as a two-dimensional array of gates $S_{i,j}$. The index j indicates the *level* of the gate, where gates closest to the inputs have the lowest values of j. (Level numbers increase from left to right in Figure 1.) For fixed j, the index i ranges over gates that are "next to" each other in the circuit, but such gates are not directly connected to each other. Each gate can get its inputs from any row (any gate in the next smaller level) in the matrix.

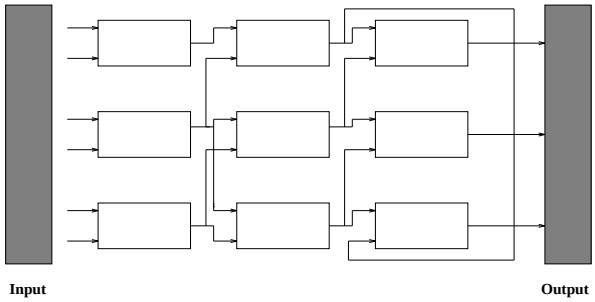

Figure 1: A gate in a two-dimensional template, gets its second input from either one of two gates in the previous column.

3.2 Example 2: A Two-Bit Multiplier

This problem is similar to the previous one, but in this case we want to design a circuit that performs the two-bit multiplication of its inputs, producing a four-bit result. The set of gates available is the same as in the previous example, and again, each gate can get its inputs from any gate in the previous level according to our matrix representation. Once more, the human designer is not required to follow these constraints. The same encoding as before is used for this circuit.

4 Using the Genetic Algorithm

The first interesting aspect of this problem is the encoding of solutions. Each circuit is encoded in the following way: $< input1 >< input2 >< gate\_type >$, where *input1* and *input2* can be any of the m previous gates or inputs at the previous column in the matrix representing the circuit (as shown in Figure 1), and *gate_type* is the gate that we will be using for that particular position of the array (either AND, NOT, OR, XOR or WIRE). A chromosome is formed with as many triplets of this kind as needed, according to the size of the matrix that the user wants to use to represent the circuit. In our experiments, both binary and floating point representation were used, and the order chosen for the gates is column-order, starting from the position $S_{1,1}$ (the top leftmost position), followed by the gate below it ($S_{2,1}$), and so on.

Since in some domains such as numerical optimization, alphabets of higher cardinality have proved to provide better results in a shorter period of time than their binary counterparts [Coello, 1996]. With this idea in mind, we decided to experiment with an alphabet of cardinality n, where n can be defined by the user and will be normally taken as the number of rows allowed in our circuit, according to the matrix encoding adopted in this problem. This representation allows the manipulation of shorter strings, it decreases the complexity of the decoding task, and it provides better solutions.

Another difficulty is the development of a good fitness function. Our approach consisted in comparing the output produced by the circuit generated by the GA with the desired values according to the truth table, on a bit-per-bit basis. Our fitness function works in two stages. At the beginning of the search, only validity of the circuit outputs is taken into account, but once functional solutions start appearing in the population, we switch to a second fitness function which tries to minimize the number of gates by rewarding those circuits that use more WIREs. For example, for the two-bit adder, we want to be able to match 48 values (only 3 bits are considered for the output, and the truth table has 16 entries), therefore a valid solution will have a fitness of 48. Whenever a valid solution is found by the GA, then the number of WIREs is added to its fitness, so if it uses, for example, 5 WIREs, its fitness will be $48 + 5 = 53$.

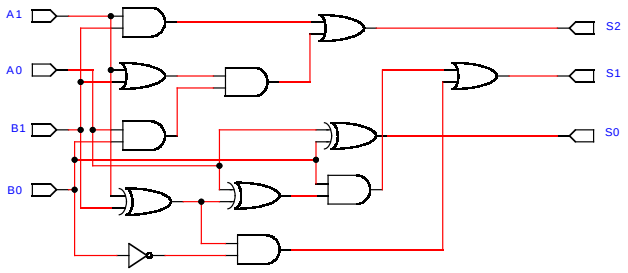

Figure 2: A 2-bit adder generated by a human designer.

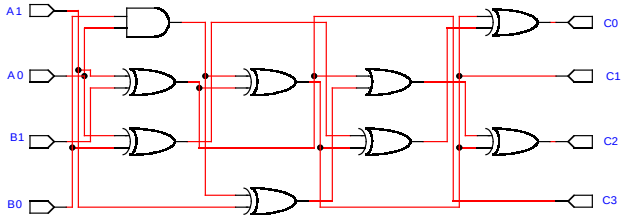

Figure 3: A sample 2-bit adder generated by our GA-based approach using binary representation.

5 Comparison of Results

We compared, for both examples, our results with those produced by an experienced human designer. Populations of 3000 chromosomes were used in all tests, and the GA was run for 200 generations.

5.1 Example 1: A Two-Bit Adder

Figure 2 shows a functional circuit generated by a human designer using Karnaugh maps.

In this example, a fitness of 48 or higher indicated a functional circuit. (There are 48 output values in the truth table for a 4-input, 3-output Boolean function.)

First, we used a traditional GA with binary representation, and the solution generated is shown in Figure 3. This circuit has only 9 gates, saving 3 from the human solution previously shown. However, we were convinced that we could do better using a higher cardinality alphabet, so we decided to experiment with a floating point representation that used the integers from 0 to 3, to represent 4 rows of inputs. The solution found was quite efficient, since only 7 gates were required (see Figure 4).

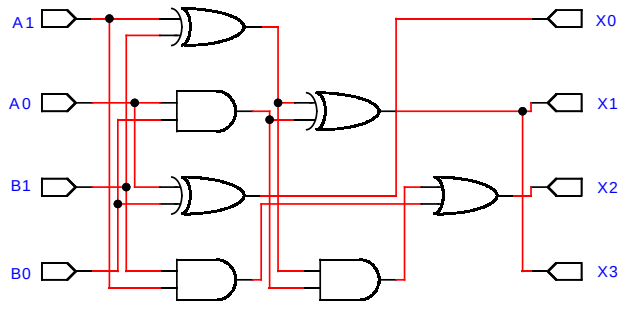

Figure 4: A sample 2-bit adder generated by our GA-based approach using floating point representation with an alphabet of cardinality 4.

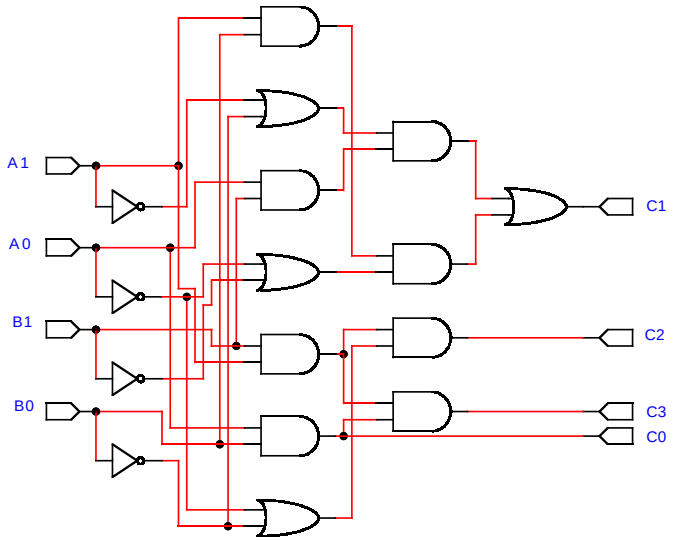

Figure 5: A 2-bit multiplier generated by a human designer.

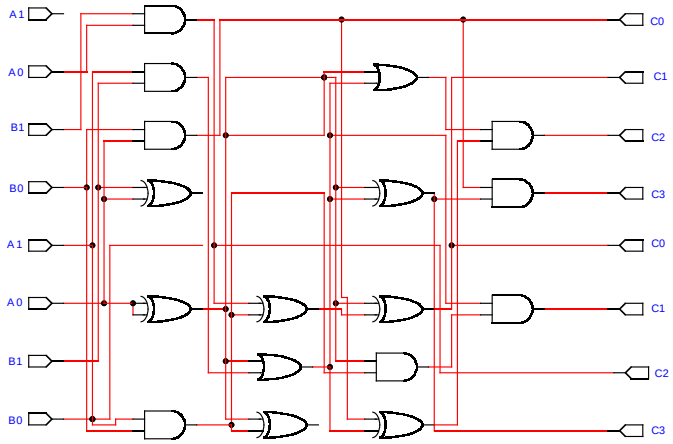

Figure 6: A 2-bit multiplier generated by a GA using binary representation.

5.2 Example 2: A Two-Bit Multiplier

This is a more complicated problem in which we compared again our solution with the circuit generated by a human designer (see Figure 5). Once more, the solution generated by the GA using binary representation was somewhat disappointing, because it uses one more gate than the solution generated by a human (see Figure 6). Then, we decided to try a GA with an alphabet of cardinality 8 (using the integers from 0 to 7), applying the modulus function to allow only 5 gates, and the result was remarkable (see Figure 7). With only 7 gates, we generated a circuit equivalent to the one produced by a human designer using 16 gates.

In this context, a fitness of 64 represents a functional circuit, and any increment above that indicates the number of WIREs used. For the GA with binary representation, the maximum fitness achieved was only 79, whereas for the GA with our alternative representation, we could find a design with a fitness of 89, as can be seen in our results.

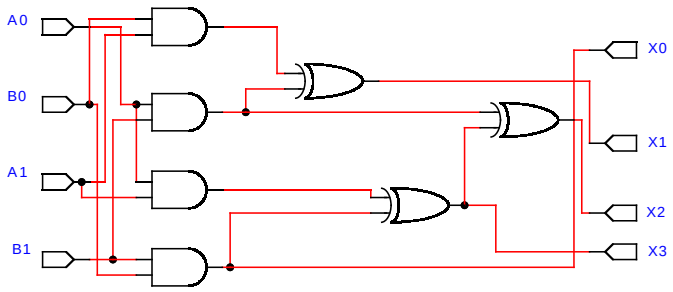

Figure 7: A 2-bit multiplier generated by a GA using an alphabet of cardinality 8.

6 Conclusions

We have demonstrated a fully implemented genetic algorithm approach to the combinational logic circuit design problem, and we have compared the results obtained by our system to circuit designs produced by experienced human designers. The results for our two simple multiple-output logic functions are quite encouraging. The automated designs use fewer gates than the human designs, and can be obtained in a few minutes by our program.

Future Work

In the future, we will determine how our GA-based approach scales to larger circuits. We are interested in building a system that can automatically design a wide range of combinational and sequential logic circuits. We are also interested in exploring optimization tradeoffs involving number of levels in the circuit, total number of gates, number of gate inputs, number of integrated circuit packages, part costs, etc. We believe that a GA-based approach has great potential to provide a practical tool for assisting designers of logic circuits.

References

[Coello, 1996] Coello, Carlos Artemio Coello 1996. *An Empirical Study of Evolutionary Techniques for Multi-objective Optimization in Engineering Design.* Ph.D. Dissertation, Department of Computer Science, Tulane University, New Orleans, LA.

[Goldberg, 1989] Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning.* Reading, Mass. : Addison-Wesley Publishing Co.

[Holland, 1992] Holland, John H. 1992. *Adaptation in Natural and Artificial Systems. An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* MIT Press, Cambridge, Massachusetts.

[Karnaugh, 1953] Karnaugh, M. 1953. A map method for synthesis of combinational logic circuits. *Transactions of the AIEE, Communications and Electronics* 72 (I):593–599.

[Koza, 1992] Koza, John R. 1992. *Genetic Programming. On the Programming of Computers by Means of Natural Selection.* The MIT Press.

[Louis, 1993] Louis, Sushil J. 1993. *Genetic Algorithms as a Computational Tool for Design.* Ph.D. Dissertation, Department of Computer Science, Indiana University.

[McCluskey, 1956] McCluskey, E. J. 1956. Minimization of boolean functions. *Bell Systems Technical Journal* 35 (5):1417–1444.

[Quine, 1955] Quine, W. V. 1955. A way to simplify truth functions. *American Mathematical Monthly* 62 (9):627–631.