Use of Evolutionary Techniques to Automate the Design of Combinational Circuits

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

† (ccoello@xalapa.lania.mx)
Laboratorio Nacional de Informática Avanzada*
Rébsamen 80, Xalapa, Veracruz 91090, México

‡({adc,hernanda}@eecs.tulane.edu)
301 Stanley Thomas Hall
Department of Computer Science
Tulane University
New Orleans, LA 70118, USA

Abstract

In this paper we propose an approach based on a genetic algorithm (GA) to design combinational logic circuits in which the objective is to minimize their total number of gates. Our results compare favorably against those produced by human designers and even another GA-based approach. We also briefly analyze the solutions found by the GA trying to find some clues on how it reduces a Boolean expression, and we indicate that such a reduction is achieved by reusing common patterns within the circuit in ways that are sometimes completely non-intuitive for a human designer. However, in small circuits, these patterns can be easier to detect and our approach could, therefore, be useful to teach circuit design since it can show students what steps to follow to simplify further a certain solution.

Keywords: circuit design, optimization, genetic algorithms, computer-aided design, circuit optimization, artificial intelligence.

IEEE classification codes: 0420 Circuits & Systems Theory, Design and Implementation, 1141 Genetic Algorithms, 1611 Design Automation, 1641 Intelligent Systems.

*Please send all correspondence to: PO Box 60326-394, Houston, Texas 77205, USA.

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.

The definition of design that fulfills the purposes of this paper is

> the process of deriving, from a specified input/output behavior, a structure (in our case a certain combination of logic gates) that is functional (produces all the outputs desired for all the inputs specified) within a certain set of specified constraints.

Furthermore, we want this design to be optimum in terms of certain structural features (e.g., the number of gates used). The design process is a very tedious and error prone task that usually requires considerable human expertise.

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 and no feedback paths.

2 PREVIOUS WORK

A general search technique inspired by natural evolution, called the *genetic algorithm* [1], has been widely used for optimization tasks [2] 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.

The design process for combinational logic circuits has evolved from its first notions [3] to a standard element of undergraduate computing curricula [4]. Standard graphical design aids such as Karnaugh Maps [5, 6] are widely used and tools suitable for computer implementation have evolved from the Quine-McCluskey Method [7, 8] to freely available tools such as Espresso [9] and MisII [10] and many commercial products.

Louis [11] is one of few sources found in the literature to address the use of GAs for the combinational logic design problem. In his dissertation [12] 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. Unfortunately, the incorporation of knowledge into the GA decreases its usefulness as a *general* search tool. Louis overcomes

this problem by defining an operator that he claims to be domain independent, but whose efficiency turns out to depend on the representation used.

Koza [13] 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. In more recent work, Koza [14, 15] has focused more towards the design of analog circuits in which the goal is to produce their appropriate topology and size so that they are functional given a certain set of components. 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. However, genetic programming produces circuits that are highly redundant and difficult to simplify automatically. Furthermore, the computer resources normally required to produce such circuits are very demanding in terms of memory and CPU time. That is why we decided to use instead a matrix representation that is encoded linearly in a chromosome, and turns out to be a compromise between the powerful tree representation used by genetic programming and the relatively weak linear representation used by a conventional genetic algorithm.

Miller et al. [16] developed (independently) an approach similar to ours, but using a more compact representation that instead of considering the inputs and gates as completely separate elements in the chromosomic string (as in our case), uses a single gene to encode a complete Boolean expression. Miller's notation does not decrease the total length of the chromosome, but it increases the cardinality of the alphabet needed, having as its main drawback the lack of flexibility of the representation to handle a larger number of inputs (the cardinality of the alphabet in Miller's case grows exponentially with respect to the number of inputs, whereas in our case, it grows linearly). Nevertheless, we will compare the results found by our approach in one example with those previously reported by Miller et al. [16].

It should be mentioned that in the work reported here, we were 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 designing a fully functional circuit from a random set of invalid circuits is a problem difficult enough as to consume most of the search time of a conventional genetic algorithm. Trying to find the feasible region in this highly constrained search space and then try to locate the optimum within such region is an even more difficult task.

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. In circuit design, one can use various criteria to define minimal-cost expressions. For example, from

a mathematical perspective, one could minimize the total number of literals or the total number of binary operations or the total number of symbols in an expression. The minimization problem is difficult for all such cost criteria. In gate networks one could minimize the total number of gates subject to such restrictions as fan-in, fan-out, number of levels, or the total number of SSI packages. In general, it is very difficult to find such minimal networks or to prove the minimality of a given network [17]. In spite of this, it is possible to solve a number of minimization problems using systematic techniques, provided that we are satisfied with less general solutions.

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* [5], which is based on a graphical representation of Boolean functions, and the *Quine-McCluskey Procedure* [7, 8], 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. Generally, several minimum functions can be obtained for a given function using either method, based on the choices made during the minimization process. All minimum functions with the same number of literals yield circuits of the same complexity; hence, any of them can be selected for implementation.

Both the Karnaugh Map and Quine-McCluskey Procedure produce *two-level* circuit forms (e.g., minimum sum of products). This is the best form if the overriding concern is minimizing propagation delay of signals through the circuit. However, in many cases a greater concern is the minimization of the number of gates present in a circuit, and a small penalty in circuit speed is acceptable. To minimize the total number of gates, it is often necessary to find a multi-level circuit form. In order to find multi-level implementations, the Karnaugh Map and Quine-McCluskey Methods must be combined with other techniques, such as algebraic manipulation of logic expressions.*

Additionally, the Quine-McCluskey Procedure is not very efficient: it can be shown that the upper bound on the number of prime implicants is $\frac{3^n}{n}$ [18], where n is the number of inputs in the truth table. This means that the CPU requirements for this procedure grows exponentially with the number of inputs. Furthermore, once the prime implicants have been found, the algorithm needs to find the minimum set cover, which is known to be an NP-complete problem [18]. Also, although some authors have proposed extensions to the

*A tool like MisII [10] *can* find multi-level forms, but requires human guidance to do so effectively.

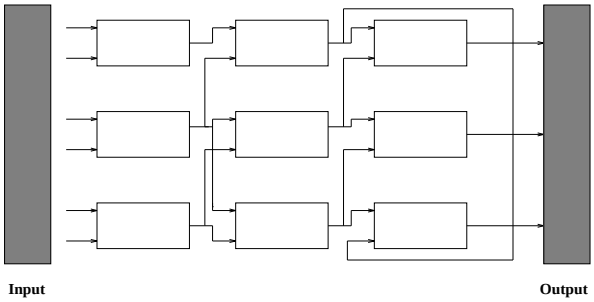

Figure 1: A gate in a two-dimensional template, gets its second input from either one of two gates in the previous column.

basic Quine-McCluskey Procedure that allow to handle XOR gates (see for example [19]), in the original proposal (which we have used here), only the basic gates are allowed (AND, OR, NOT), and a human designer has to perform further refinements in order to introduce XOR gates into the circuit.

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 two human designers: one who used Karnaugh maps and another who used the Quine-McCluskey Procedure (unless indicated otherwise in the examples). 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 and the Quine-McCluskey Procedure do not support the identification of XOR terms as well as they support "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.

4 USING THE GENETIC ALGORITHM

The first interesting aspect of this problem is the encoding of solutions as chromosomic strings that the GA can evolve. The representation chosen for our work is a bidimensional matrix as the one suggested by Louis [11] in which each matrix element is a gate (there are 5 types of gates: AND, NOT, OR, XOR and WIRE) that receives its 2 inputs from any

Input 1	Input 2	Gate Type

Figure 2: Encoding used for each of the matrix elements that represent a circuit.

gate[†] at the previous column as shown in Figure 1. More formally, we can say that any circuit can be represented as a bidimensional array of gates $S_{i,j}$, where j indicates the *level* of a gate, so that those gates closer to the inputs have lower values of j. (Level values are incremented from left to right in Figure 1). For a fixed j, the index i varies with respect to the gates that are "next" to each other in the circuit, but without being necessarily connected. It is interesting to notice that if a row-order encoding is used, the problem becomes disruptive [11], making it very hard for the GA. The reason is that using such an encoding, any circuit designs that are close in two-dimensional (phenotypic) space may be far apart in one-dimensional (genotypic) space, making it difficult to preserve highly fit schemas (in GA terminology, we say that the problem is deceptive [20]).

A chromosomic string encodes the matrix shown in Figure 1 by using triplets in which the 2 first elements refer to each of the inputs used, and the third is the corresponding gate as shown in Figure 2 (only 2-input gates were used in this work). For the gates at the first level (or column), the possible inputs for each gate were those defined by the truth table given by the user (a modulo function was implemented to allow more rows than allowable inputs).

Our goal was then to produce a fully functional design (i.e., one that produces all the expected outputs for any combination of inputs according to the truth table given for the problem) which maximizes the number of WIREs[‡].

A critical part of getting a GA approach to succeed in this problem has to do with the representation scheme used by the genetic algorithm. Although it has been argued that a binary representation provides the maximum number of schemata [21] it turns out that 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 [22]. 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 as has been shown in previous work, it provides better solutions than its binary counterpart [23, 24].

Another difficulty is the development of a good fitness function. Again, our initial

[†]It is worth mentioning that Louis fixes the position of one of the inputs to reduce the size of the search space [12].

[‡]WIRE basically indicates a null operation, or in other words, the absence of gate, and it is used just to keep regularity in the representation used by the GA that otherwise would have to use variable-length strings.

approach was to use a slight variation of the function suggested by Louis in his dissertation [12], which consists of the number of correct operations performed. Instead of checking on the possible outcomes for each of the possible combinations of inputs, we checked only the final result on each case, requiring, fewer comparisons than in Louis' approach. However, we found in further tests that it is in general more convenient to check the output on a bit-per-bit basis, because that provides more information to the GA to guide the search, and better results can be achieved in less time. So, to evaluate the fitness of each chromosome, we looped through all the possible values for each of the inputs and computed the number of hits that the encoded solution was able to achieve with respect to the desired outputs (i.e., the outputs defined in the truth table).

Our fitness function works in two stages. At the beginning of the search, only validity of the circuit outputs is taken into account, and the GA is basically exploring the search space. Once a functional solution appears, then the fitness function is modified such that any valid designs produced are rewarded for each WIRE gate that they include, so that the GA tries to find the circuit with the maximum number of WIREs that performs the function required. It is at this stage that the GA is actually exploiting the search space, trying to optimize the solutions found (in terms of their number of gates) as much as possible.

It should be mentioned that although at first sight the size of the search space for some instances of this problem may seem too small to even attempt to use a heuristic function, that is not true. For the representation used for this work, if we assume a cardinality n and a chromosomic lenght l, the size of the intrinsic search space is n^l. Both the cardinality and the length of a string depend on the size of the matrix used to solve the circuit: $l = 3 \times t$, where $t = r \times q$, and r and q are the number of rows and columns of the matrix respectively. As we will see in a further section, even for small circuits, the size of the intrinsic search space is sufficiently large as to make impossible to enumerate all the possible solutions to a circuit in a reasonable amount of time.

5 COMPARISON OF RESULTS

We have used several circuits of different degrees of complexity to test our approach. For the purposes of this paper, 5 examples were chosen to illustrate our approach, and the results produced with the GA were compared with those generated by human designers and, in one case, with another GA-based approach.

In each case, the size of the matrix used to fit the circuit was determined using the following procedure:

1. Start with a square matrix of size 5.
2. If no feasible solution is found using this matrix, then increase the number of columns by one.
3. If no feasible solution is found using this matrix, then increase the number of rows by one.

4. Repeat steps 2 and 3 until a suitable matrix is produced.

As we will see in the following examples, it was normally the case that for small circuits a matrix of 5×5 was sufficient. However, in our last example, it was necessary to reach a matrix size of 6×7. This made necessary to run the GA for more generations, performing, in consequence, more fitness function evaluations. This situation normally arises with circuits having several outputs, although in some cases, such as in the 2-bit multiplier of our fourth example, even a 5×5 matrix may be enough to find the best known circuit. This procedure is not very efficient but it serves as a rudimentary alternative to the use of truly variable-length chromosomes, which is a choice that we are currently exploring.

The other issue is regarding the crossover and mutation rates. After a series of experiments, we decided to use two-point crossover with a probability of 50% and uniform bit mutation with a probability that would guarantee that each string would have a 50% chance of being mutated in at least one position across its length. Since mutation was applied on a single-gene basis, we used as our probability of mutation the result of dividing this 50% by the length of the string. Since the length of the strings used to solve the first four examples is 75, the probability of mutation in those cases was 0.006667. The last example used a longer string (126), which made necessary to use a lower probability of mutation (0.003968). Binary tournament selection with full generational replacement was used in all cases and the termination criterion adopted was a maximum number of generations defined by the user. We decided not to adopt a termination criterion based on the lack of improvement of a certain solution after a number of generations, because we found it difficult to define a threshold that could ensure no further improvement of a solution. In some circuits, we found that there would be no improvement of a solution after a relatively large number of generations and then the GA would be able to jump abruptly to a much better solution. However, this issue deserves further attention.

In all the examples we kept the best individual of each generation (elitism). The maximum number of generations was arbitrarily set to a reasonably large number, and the population size was chosen based on a number of independent runs. For the case of the first four examples, the population size was ranged from 100 to 3000 individuals with increments of 100 (30 runs) and the maximum number of generations was set to 300. In the case of the last example, the population size was ranged from 1000 to 3000 with increments of 100 (20 runs), and the maximum number of generations was set to 2000. In each case, the value shown for the population size is the one that produced the best results for that particular circuit (whenever the optimum solution was found with several population sizes, we reported the results produced with the smallest of them).

5.1 Example 1

Our first example has 3 inputs and one output, as shown in Table 1. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$). The cardinality c used for this problem was $max(r, g)$, where g refers to the

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Table 1: Truth table for the circuit of the first example.

number of allowable gates (since only the inputs from the previous level are considered, the number of columns does not affect the cardinality used by the GA). Since $g = 5$, and $c = 5$, then the size of the intrinsic search space for this problem is $c^l = 5^{75} \approx 2.6 \times 10^{52}$.

The comparison of the results produced by the GA, and two human designers are shown in Table 2. In this and all the further examples, designer 1 used Karnaugh Maps plus Boolean algebra identities to simplify the circuit, whereas designer 2 used the Quine-McCluskey Procedure.

Notice that in this case, if we develop the expression produced by human designer 1, we can actually find a solution with only 4 gates: $Z(X \oplus Y) + Y(X \oplus Z) = ZX \oplus ZY + YX \oplus YZ$. We can now rearrange terms: $(ZX + YX) \oplus (YZ)$ and finally, we can take the common term X outside: $X(Z + Y) \oplus (YZ)$. This solution has only 4 gates, but is not exactly the same found by the GA. It is interesting to notice how the solution produced by the GA can certainly motivate to explore further our own solution produced by hand and to apply certain algebraic transformations that will bring us closer to the optimum found by the GA. This is particularly useful for teaching purposes, since it can show students how to apply some non-obvious algebraic transformations to a Boolean expression to reduce it more. However, these patterns can be clearly seen only in small circuits, since in larger circuits they become so complex that it becomes very hard to derive them by hand.

The parameters used by the GA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 300, maximum number of generations = 300. Convergence to the optimum was achieved in generation 81.

5.2 Example 2

Our second example has 4 inputs and one output, as shown in Table 3. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$). The size of the intrinsic search space for this problem is then $c^l = 5^{75} \approx 2.6 \times 10^{52}$.

The comparison of the results produced by the GA, a human designer, and Sasao's

Genetic Algorithm	**Human Designer 1**	**Human Designer 2**
$F = Z(X + Y) \oplus (XY)$	$F = Z(X \oplus Y) + Y(X \oplus Z)$	$F = X'YZ + X(Y \oplus Z)$
4 gates	5 gates	6 gates
2 ANDs, 1 OR, 1 XOR	2 ANDs, 1 OR, 2 XORs	3 ANDs, 1 OR, 1 XOR, 1 NOT

Table 2: Comparison of results between our genetic algorithm and 2 human designers for the circuit of the first example.

Z	**W**	**X**	**Y**	**F**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Table 3: Truth table for the circuit of the second example.

Genetic Algorithm
$F = (WYX' \oplus ((W + Y) \oplus Z \oplus (X + Y + Z)))'$
10 gates
2 ANDs, 3 ORs, 3 XORs, 2 NOTs
Human Designer 1
$F = ((Z'X) \oplus (Y'W')) + ((X'Y)(Z \oplus W'))$
11 gates
4 ANDs, 1 OR, 2 XORs, 4 NOTs
Sasao
$F = X' \oplus Y'W' \oplus XY'Z' \oplus X'Y'W$
12 gates
3 XORs, 5 ANDs, 4 NOTs

Table 4: Results produced by our GA, a human designer and Sasao's technique for the second example.

approach [25] are shown in Table 4. Sasao has used this circuit to illustrate his circuit simplification technique based on the use of ANDs & XORs. His solution uses, however, more gates than the circuit produced by the GA which is rather atypical, because it negates the whole Boolean expression. This, however, turns out to save one gate with respect to the best solution produced by a human designer, although the two solutions do not look alike at all.

The parameters used by the GA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 1000, maximum number of generations = 300. Convergence to the optimum was achieved in generation 99.

5.3 Example 3

Our third example has 4 inputs and one output, as shown in Table 5. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$). The size of the intrinsic search space for this problem is also $5^{75} \approx 2.6 \times 10^{52}$.

This is an example in which is not easy to go from the final expression produced by a human designer to the solution found by the GA. Even when the solution produced by human designer 1 has some resemblance with the one produced by the GA, it does not seem obvious what Boolean identities to use to transform the solution by hand into the expression produced by the GA.

The parameters used by the GA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 2000, maximum number of generations = 300. Convergence to the optimum was achieved in generation 155.

The comparison of the results produced by the GA and two human designers are shown

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

Table 5: Truth table for the circuit of the third example.

in Table 6.

5.4 Example 4

Our fourth example has 4 inputs and 4 outputs, and it's a 2-bit multiplier as shown in Table 7. In this case, the matrix used was of size 5×5, and the chromosomic length was 75 ($r = 5, q = 5, t = 5 \times 5 = 25, l = 3 \times t = 75$). The size of the intrinsic search space for this problem is (as in previous examples) $5^{75} \approx 2.6 \times 10^{52}$.

The comparison of the results produced by the GA, a human designer, and Miller et al. [16] are shown in Table 8. It should be mentioned that Miller et al. consider their solution to contain only 7 gates because of the way in which they encoded their Boolean functions (the reason is that they encoded NAND gates which is also valid in practice). However, since we considered each gate as a separate chromosomic element, we count each of them, including NOTs that are associated with AND & OR gates. Regardless of that fact, it is more important to point out that Miller et al. found their solution performing 50 runs of 3,000,000 evaluations each, whereas in our case, we only performed 30 runs of 600,000 evaluations each.

Notice that the only difference between the solution produced by human designer 1 and the GA is on the output C_2. This is the sort of example in which the solution may seem difficult to achieve by a human designer, because by looking at the solution for C_2 produced by the GA, one could think that is more inefficient. However, the GA is actually reusing

Genetic Algorithm
$F = ((A \oplus B) \oplus AD) + (C + (A \oplus D))'$
7 gates
1 AND, 2 ORs, 3 XORs, 1 NOT
Human Designer 1
$F = ((A \oplus B) \oplus ((AD)(B + C))) + ((A + C) + D)'$
9 gates
2 ANDs, 4 ORs, 2 XORs, 1 NOT
Human Designer 2
$F = A'B + A(B'D' + C'D)$
10 gates
4 ANDs, 2 ORs, 4 NOTs

Table 6: Results produced by our GA and 2 human designers for the third example.

$\mathbf{A}_1$	$\mathbf{A}_0$	$\mathbf{B}_1$	$\mathbf{B}_0$	$\mathbf{C}_3$	$\mathbf{C}_2$	$\mathbf{C}_1$	$\mathbf{C}_0$
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Table 7: Truth table for the 2-bit multiplier of the fourth example.

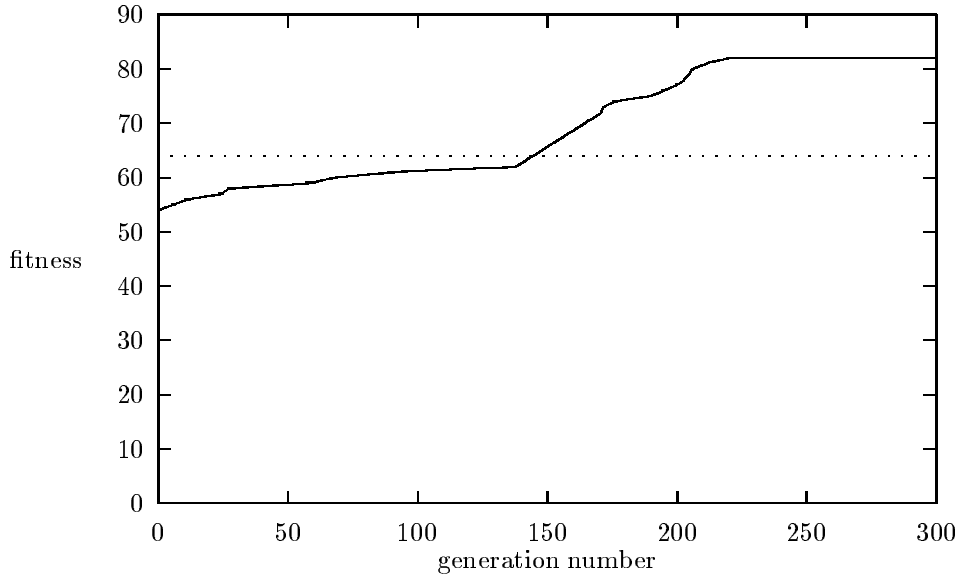

Figure 3: Convergence graph of the GA used to solve the fourth example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

gates which, in terms of the overall circuit, turns out to be more efficient, because it saves one gate with respect to the best solution produced by a human designer. With respect to Miller's solution, notice that it uses the same value for C_2 as human designer 1, but it has a much more complex expression for C_3. That is the reason why their overall circuit uses two more gates than our solution.

The parameters used by the GA for this example are the following: crossover rate = 0.5, mutation rate = 0.007, population size = 2000, maximum number of generations = 300. The convergence graph of the GA is shown in Figure 5.4. Convergence to the optimum was achieved in generation 220.

5.5 Example 5

Our fifth example has 4 inputs and 3 outputs, as shown in Table 9. In this case, the matrix used was of size 6×7, and the chromosomic length was 126 ($r = 6, q = 7, t = 6 \times 7 = 42, l = 3 \times t = 126$). The cardinality $c = max(r, g) = 6$ The size of the intrinsic search space for this problem is $c^l = 6^{126} \approx 1.1 \times 10^{98}$.

The comparison of the results produced by the GA and a human designer are shown in Table 10.

This is an extreme case of how the GA can reuse blocks of the circuit to optimize the total number of gates. Notice how the functions produced by the GA for each separated

Genetic Algorithm	**Human Designer 1**
$C_0 = A_0B_0$	$C_0 = A_0B_0$
$C_1 = A_0B_1 \oplus A_1B_0$	$C_1 = A_0B_1 \oplus A_1B_0$
$C_2 = A_1B_1 \oplus (A_0B_0A_1B_1)$	$C_2 = A_1B_1(A_0B_0)'$
$C_3 = A_0B_0A_1B_1$	$C_3 = A_1A_0B_1B_0$
7 gates	8 gates
5 ANDs, 2 XORs	6 ANDs, 1 XORs, 1 NOT
Human Designer 2	**Miller et al.**
$C_0 = A_0B_0$	$C_0 = A_0B_0$
$C_1 = (B_1 + B_0)(A_1 + A_0)((A_1A_0) \oplus (B_1B_0))$	$C_1 = A_1B_0 \oplus A_0B_1$
$C_2 = A_1B_1(A_0B_0)'$	$C_2 = (A_0B_0)'(A_1B_1)$
$C_3 = A_1B_1A_0B_0$	$C_3 = (A_1B_0 \oplus A_0B_1)'(A_1B_0)$
12 gates	9 gates
8 ANDs, 1 XOR, 2 ORs, 1 NOT	6 ANDs, 1 XORs, 2 NOTs

Table 8: Results produced by our GA, 2 human designers and Miller et al. for the circuit of the fourth example.

A	**B**	**C**	**D**	**F1**	**F2**	**F3**
0	0	0	0	1	0	0
0	0	0	1	0	1	0
0	0	1	0	0	1	0
0	0	1	1	0	1	0
0	1	0	0	0	0	1
0	1	0	1	1	0	0
0	1	1	0	0	1	0
0	1	1	1	0	1	0
1	0	0	0	0	0	1
1	0	0	1	0	0	1
1	0	1	0	1	0	0
1	0	1	1	0	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1
1	1	1	1	1	0	0

Table 9: Truth table for the circuit of the fifth example.

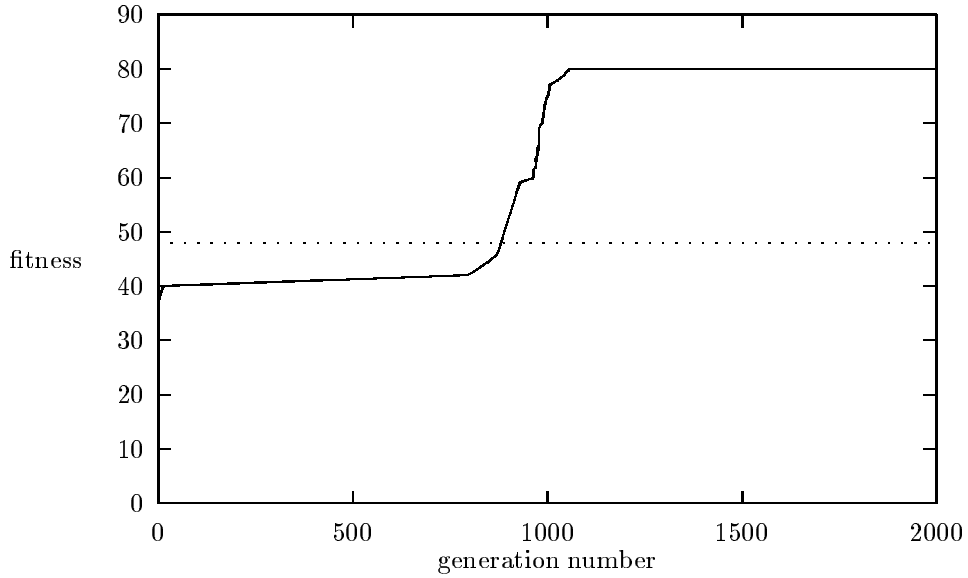

Figure 4: Convergence graph of the GA used to solve the fifth example. The feasibility barrier is indicated with a horizontal line (any value above it represents a fully functional circuit).

output are more complex, but since they use common blocks, the total number of gates is almost half of what one of the human designers required.

The parameters used by the GA for this example are the following: crossover rate = 0.5, mutation rate = 0.004, population size = 2800, maximum number of generations = 2000. The convergence graph of the GA is shown in Figure 5.5. Convergence to the optimum was achieved in generation 1059.

6 DISCUSSION OF RESULTS

It is interesting to notice that the GA tends to favor the use of XOR gates, since this gate allows to produce in many cases solutions with a shorter symbolic representation. These solutions, however, are not entirely obvious for a human designer who can normally visualize easily only designs with the basic gates (AND, OR, NOT) and with XORs that are not nested. The GA, on the other hand, tends to use very often nested XORs to produce the same effect that a human designer would achieve combining the basic gates. There it lies the main reason for which the GA tends to produce circuits that are difficult for a human to design and even to understand.

Because of the different population sizes used in our experiments, we considered important to do a sensitivity analysis of the GA to this parameter. In our experiment, we ran the

Genetic Algorithm
$F1 = ((A \oplus C) + (B \oplus D))'$
$F2 = ((B \oplus D) + (A \oplus C))(((A \oplus C)A \oplus (D + (A \oplus C))) + ((B \oplus D) + (A \oplus C))'$
$F3 = (((A \oplus C)A \oplus ((A \oplus C) + D)) + ((B \oplus D) + (A \oplus C))')'$
10 gates
3 XORs, 3 ORs, 2 ANDs, 2 NOTs
Human Designer 1
$F1 = (A \oplus C)'(B \oplus D)'$
$F2 = B'D(A' + C) + A'C$
$F3 = BD'(A + C') + AC'$
19 gates
2 XORs, 4 ORs, 7 ANDs, 6 NOTs
Human Designer 2
$F1 = (A \oplus C)'(B \oplus D)'$
$F2 = A'C + (A \oplus C)'(B'D)$
$F3 = (F1 + F2)'$
13 gates
2 XORs, 2 ORs, 4 ANDs, 5 NOTs

Table 10: Results produced by our GA and 2 human designers for the fifth example.

same program using population sizes that went from 100 to 3000 in increments of 100 (a total of 30 runs) and computed the average fitness and the standard deviation of the results produced from these runs. For the first example, we found an average fitness of 28.73 (the optimum solution had a fitness of 29), with a standard deviation of 0.82768. Only in 3 of these 30 runs, the GA was not able to converge to the optimum solution after 300 generations. This, however, was not always the case. For the second example, the average fitness was 19.33 (the optimum had a fitness of 31), with a standard deviation of 7.2793, and the GA was able to converge to the optimum only in 6 cases (out of 30). A similar behavior was found in the third and fourth examples, since the average fitness was 22 (the optimum had a fitness of 34) and 67.8 (the optimum had a fitness of 82), respectively. For the third example, the standard deviation was 7.6654, and for the fourth, it was 8.384. For the third example, the GA was able to converge to the optimum only in 9 runs, and for the fourth example, it could converge only in 5 occasions (out of 30). Finally, for the fifth example, the average fitness was 63.8 with a standard deviation of 13.3685. In this example, the GA was able to converge to the optimum solution in only 2 runs (out of 20). Although we find it difficult to derive any rules regarding the population size that seem more appropriate in general, we can say that as we increase the size of the population, the GA tends to find the global optimum more easily (but possible in a larger number of generations). Therefore, when nothing is known about the problem at hand (within this specific domain), it might be a good idea to start with a relatively large population size (2000 chromosomes at least).

Example No.	Time (seconds)
1	43
2	179
3	551
4	1090
5	8890

Table 11: CPU times required for one run of each of the examples presented in this paper (the program was written in C under Unix and run in a Silicon Graphics workstation with 64 Mbytes of memory and a 134 MHz processor).

Our second interest was to determine the sentitivity of the GA to the random number seed. For that sake, we ran the same program corresponding to the first example 30 times using different random number seeds. The average fitness found was 28.6, with a standard deviation of 0.4889. Only in 3 cases the GA was not able to find the optimum solution after 300 generations. A similar behavior was found with the other examples, showing a low dependency of the GA performance on the initial random distribution of the population.

The use of a relatively low crossover rate (50%) could be suspicious for some people who could argue that the use of pure mutation could be enough in this problem. We decided to try the same examples using only mutation. The results were interesting. For instance, for the second example the average fitness was slightly higher than before (19.67), but the GA was not able to converge to the optimum in any of the 30 runs performed. As we increase the complexity of the circuit, the use of pure mutation turns out to be less efective. For example, when applied to the fourth example, the average fitness was 64.033 (lower than when using crossover) and the GA was not able to converge to the optimum in any of the 30 runs performed. A similar behavior was found in the other examples (except for the first one in which the global optimum was achieved), which seems to indicate that the GA with pure mutation can be useful to approach relatively fast the feasible region in this domain, but needs the explotation capabilities of the crossover operator to achieve the global optimum.

Although it may be argued that the relatively high number of evaluations performed by the GA is far beyond the search capabilities of a human designer, it must be said that the GA is in fact exploring a minimum portion of the total search space that is intractable by simple brute force search methods. For instance, for the examples in which the size of the intrinsic search space is 2.6×10^{52}, even assuming that our computer could evaluate 1×10^{12} solutions per second, we would need 8.39×10^{32} years to explore the entire search space using a brute force approach.

Finally, one issue that deserves attention is the scalability of the approach to real-world circuits. The approach presented in this paper can easily exceed the memory capabilities and processor speed of any workstation when applied to larger circuits unless their outputs are considered separately. To have an idea of the computing capabilities required, consider

the CPU times indicated in Table 11 which correspond to one run of each of the examples presented in this paper (using the parameters that provided the best solution for each example as indicated before). It is important to mention that to find the best solution to each example, we really ran each GA 30 times, varying the population size from 100 to 3000. Each full set of runs took between 18 and 29 hours to complete, depending on the network load (all the runs were performed simultaneously). In general, it should be obvious to see that the time required for each run tends to increase as we solve more complex circuits. This is an intrinsic limitation of the fixed-length representation adopted in this research, but we are exploring the use of more powerful representations (i.e., trees) that can overcome this limitation and allow the solution of larger circuits in a reasonable amount of time [26]. So far, our approach is limited to circuits of up to 64 entries and 3 outputs in their truth table. In any case, evolvable hardware in general is currently limited by the computing power available and it is not very difficult to state problems that exceed such capabilities.

7 CONCLUSIONS

We have shown a technique to design combinational logic circuits using a genetic algorithm. Our GA has been able to find circuits that are smaller (in terms of the total number of gates) than those produced by human designers and even another GA-based approaches, performing a relatively small number of evaluations with respect to the total size of the search space.

By analyzing the solutions produced by the GA, we can see how it reuses components within a circuit as to reduce the total number of gates, even if in the process the Boolean expression for a certain output could become more complex than the one produced by a human designer. This reuse of components seems to be the key to find simplified versions of a circuit, but it becomes harder to understand it as we increase the complexity of the circuit. However, for small circuits, the use of our GA-based approach could be useful to teach how to employ certain DeMorgan laws or any other Boolean simplification techniques, since the solutions produced in those cases are normally easily traceable by hand.

8 FUTURE WORK

It is important to realize the difficulties of the GA to even generate a feasible circuit in early generations. If proper matrix dimensions are not provided, the GA may not converge at all regardless of the parameters used.

Although we have provided some basic guidelines to deal with this problem, we want to explore more flexible representations that allow an easier encoding of variable-length Boolean expressions as to minimize the tune up required to design any sort of combinational circuit. We would also like to extend our representation to handle circuits with more than two inputs. Right now, the most promising alternative seems to be genetic programming

[13], but we still have to define a way of optimizing the Boolean expressions produced which are, generally, quite long.

In the future we would like to consider also other factors in our fitness functions, such as the number of levels, the number of packages of integrated circuits used, the costs of each component, etc.

Later on, we would like to move towards more complex circuits (for example sequential), and consider more complex factors such as time delays. Also, it would be desirable to build a graphical interface to our system, which currently has a text-based interface that makes the interpretation of results a little bit difficult for those not familiar with the software.

ACKNOWLEDGMENTS

The authors would like to thank the two anonymous reviewers for their valuable comments that helped them improve this paper. The first author also acknowledges the support received from CONACyT through project number I-29870 A.

References

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

[2] Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning* (Addison-Wesley Publishing, Reading, Massachusetts).

[3] Shannon, C. E. (1938) A symbolic analysis of relay and switching circuits, *Transactions of the AIEE*, **57**, pp. 713–723.

[4] Roth, C. H.,Jr. (1992) *Fundamentals of Logic Design (4th Edition)* (West Publishing Company, Minneapolis).

[5] Karnaugh, M. (1953) A map method for synthesis of combinational logic circuits, *Transactions of the AIEE, Communications and Electronics*, **72**, No. I, pp. 593–599.

[6] Veitch, E. W. (1952) A chart method for simplifying boolean functions, *Proceedings of the ACM*, May, pp. 127–133.

[7] Quine, W. V. (1955) A way to simplify truth functions, *American Mathematical Monthly*, **62**, No. 9, pp. 627–631.

[8] McCluskey, E. J. (1956) Minimization of boolean functions, *Bell Systems Technical Journal*, **35**, No. 5, pp. 1417–1444.

[9] Brayton, R. K., Hachtel, G. D., McMullen, C. T., and Sangiovanni-Vincentelli, A. L. (1984) *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, Dordrecht, The Netherlands).

[10] Brayton, R. K., Rudell, R., Sangiovanni-Vincentelli, A., and Wang, A. R. (1987) MIS: A multiple-level logic optimization system, *IEEE Transactions on Computer-Aided Design*, **CAD-6**, No. 6, pp. 1062–1081.

[11] Louis, S. and Rawlins, G. Designer genetic algorithms: Genetic algorithms in structure design, In Belew, R. K. and Booker, L. B. (eds) (1991) *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 53–60, San Mateo, California, Morgan Kaufmann Publishers.

[12] Louis, S. J. (1993) *Genetic Algorithms as a Computational Tool for Design*, PhD thesis, Department of Computer Science, Indiana University.

[13] Koza, J. R. (1992) *Genetic Programming. On the Programming of Computers by Means of Natural Selection* (The MIT Press, Cambridge, Massachusetts).

[14] Koza, J. R., Forrest H. Bennett, I., Andre, D., and Keane, M. A. Automated WYWIWYG design of both the topology and component values of electrical circuits using genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (eds) (1996) *Proceedings of the First Annual Conference on Genetic Programming*, pp. 123–131, Cambridge, Masachussetts, The MIT Press.

[15] Koza, J. R., Andre, D., Forrest H. Bennett, I., and Keane, M. A. (1996) Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. (eds), *Proceedings of the First Annual Conference on Genetic Programming*, pp. 132–140, Cambridge, Masachussetts, The MIT Press.

[16] Miller, J. F., Thomson, P., and Fogarty, T. , Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, In Quagliarella, D., Périaux, J., Poloni, C., and Winter, G. (eds) (1998) *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pp. 105–131. Morgan Kaufmann, Chichester, England.

[17] Brzozowski, J. A. and Yoeli, M. (1976) *Digital Networks* (Prentice Hall, Englewood Cliffs, New Jersey).

[18] Katz, R. H. (1994) *Contemporary logic design* (Benjamin/Cummings Publishing Co., Redwood City, California).

[19] Turton, B. C. H. (1996) Extending Quine-McCluskey for Exclusive-Or Logic Synthesis. *IEEE Transactions on Education*, **39**, No. 1, pp. 81–85.

[20] Grefenstette, J. J. Deception Considered Harmful, In Whitley, L. D., (ed) (1993) *Foundations of Genetic Algorithms 2*, pp. 75–91, Morgan Kaufmann, San Mateo, California.

[21] Michalewicz, Z. (1992) *Genetic Algorithms + Data Structures = Evolution Programs*, (Springer-Verlag, New York, New York).

[22] Coello Coello, C. A. (1996) *An Empirical Study of Evolutionary Techniques for Multiobjective Optimization in Engineering Design*, PhD thesis, Department of Computer Science, Tulane University, New Orleans, Louisiana.

[23] Coello Coello, C. A., Christiansen, A. D., and Hernández Aguirre, A. Using genetic algorithms to design combinational logic circuits, In Dagli, C. H., Akay, M., Chen, C. L. P., Fernández , B. R., and Ghosh, J. (eds) (1996) *Intelligent Engineering Systems Through Artificial Neural Networks. Volume 6. Fuzzy Logic and Evolutionary Programming*, pp. 391–396, ASME Press, St. Louis, Missouri, USA.

[24] Coello Coello, C. A., Christiansen, A. D., and Hernández Aguirre, A. Automated Design of Combinational Logic Circuits using Genetic Algorithms. In Smith, D. G., Steele, N. C., and Albrecht, R. F. (eds) (1997) *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pp. 335–338. Springer-Verlag, University of East Anglia, England.

[25] Sasao, T., (ed) (1993) *Logic Synthesis and Optimization* (Kluwer Academic Publishers, Dordrecht, The Netherlands).

[26] Hernández Aguirre, A., Coello Coello, Carlos A., and Buckles, Bill P. A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers, In Stoica, A., Keymeulen, D., and Lohn, J., (eds) (1999) *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, pp. 46–53, Los Alamitos, California, IEEE Computer Society Press.

APPENDIX A: CIRCUITS PRODUCED

In this appendix, we include the graphical representation of the optimum circuits generated by our GA-based approach. The symbols used in these pictures to represent the gates of a circuit are shown in Figure 5.

The graphical representation of the optimum circuits produced by the GA for the five examples included in this paper are shown in Figures 6, 7, 8, 9, and 10.

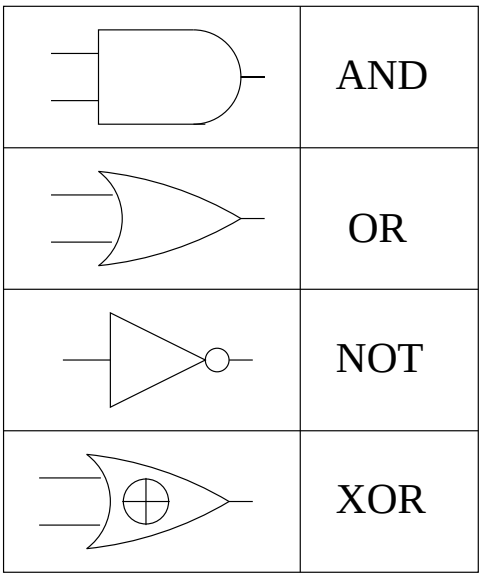

Figure 5: Symbols used to represent the gates of the circuits included in this paper.

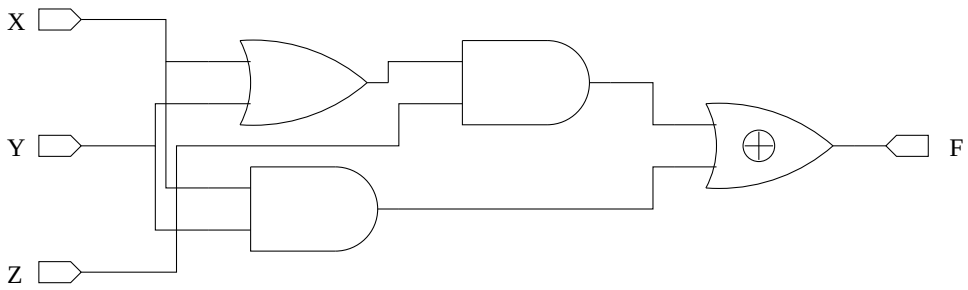

Figure 6: Circuit produced by our GA for the first example.

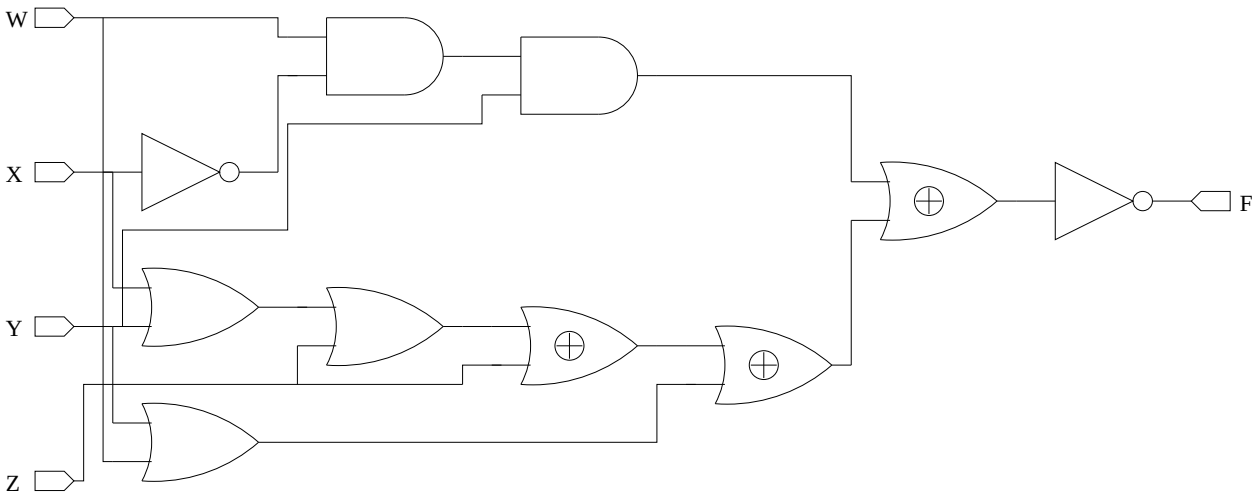

Figure 7: Circuit produced by our GA for the second example.

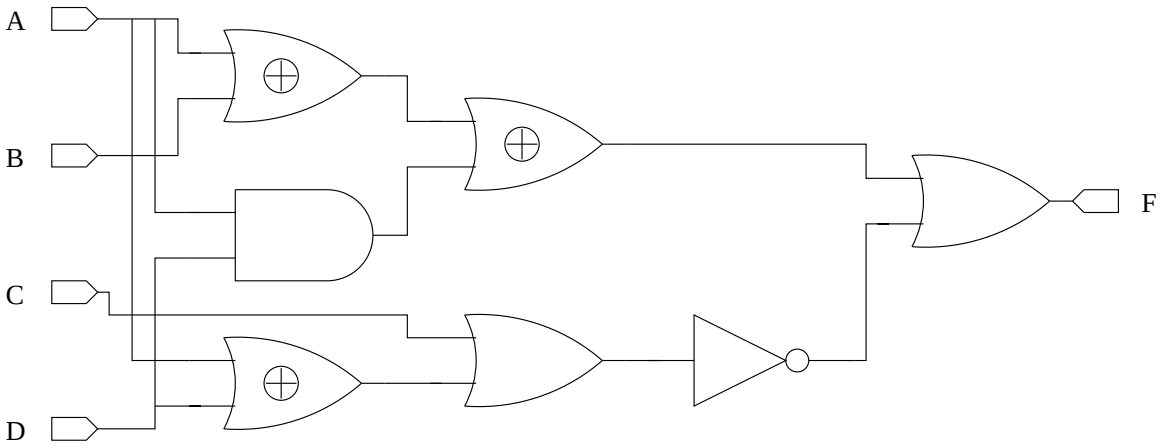

Figure 8: Circuit produced by our GA for the third example.

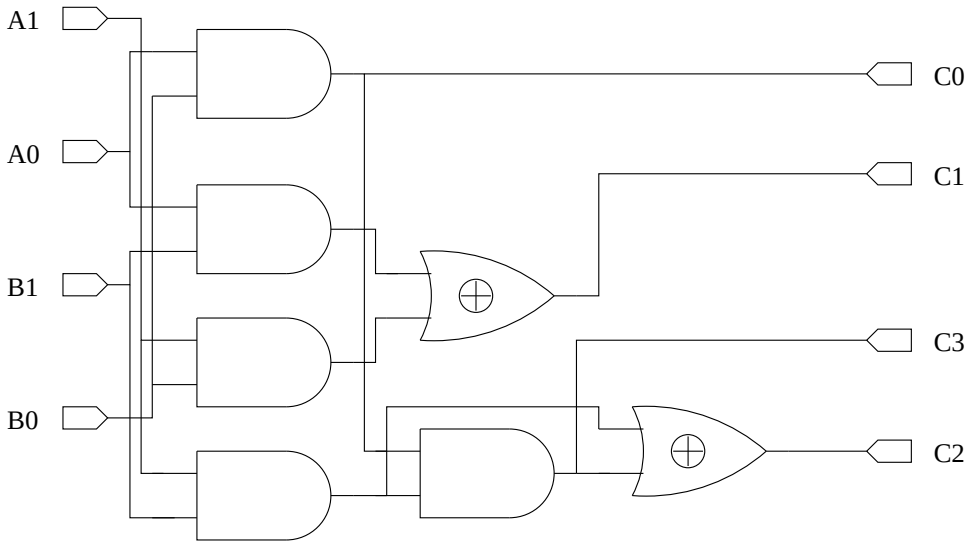

Figure 9: Circuit produced by our GA for the fourth example.

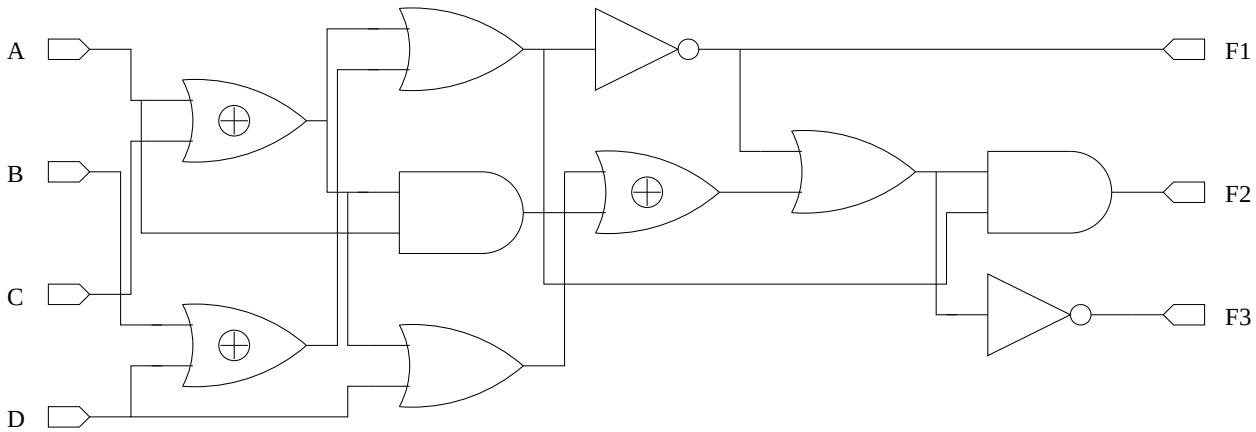

Figure 10: Circuit produced by our GA for the fifth example.