A Genetic Programming Approach to Logic Function Synthesis by means of Multiplexers

Arturo Hernández Aguirre
211 Stanley Thomas Hall
Department of Computer Science, Tulane University
New Orleans, LA 70118, USA
hernanda@eecs.tulane.edu

Carlos A. Coello Coello
Laboratorio Nacional de Informática Avanzada
Rébsamen 80, A.P. 696
Xalapa, Veracruz, México 91090
ccoello@xalapa.lania.mx

Bill P. Buckles
211 Stanley Thomas Hall
Department of Computer Science, Tulane University
New Orleans, LA 70118, USA
buckles@eecs.tulane.edu

Abstract

This paper presents an approach based on the use of genetic programming to synthesize logic functions. The proposed approach uses the 1*-control line multiplexer as the only design unit, defining any logic function (defined by a truth table) through the replication of this single unit. Our fitness function first explores the search space trying to find a feasible design and then concentrates in the minimization of such (fully feasible) circuit. The proposed approach is illustrated using several sample Boolean functions.*

1. Introduction

Although evolvable hardware is nowadays a very popular area of research, the use of evolutionary techniques to optimize combinational circuits at the gate-level has been approached by only a few researchers [12, 6, 3, 9, 8].

In the past, we have concentrated on the optimum design of combinational circuits at the gate level using genetic algorithms (GAs) [2, 3]. In the past, we have used the number of gates as the design metric to be minimized in a circuit, so that our goal was to produce fully functional circuits that required the smallest possible number of gates (chosen from a certain set defined by the user).

However, the use of this metric may not be realistic in VLSI systems design where the emphasis is to decrease the whole manufacturing cost rather than reducing the total number of components used [4]. It is common, therefore, to replicate the same unit as many times as possible, although this may lead to circuits with a larger number of gates. Furthermore, in this domain, the silicon surface needed to implement any logical component is another very important factor that deserves consideration, and our previous work had not considered either of these issues.

Our interest in VLSI systems design led us to restate the circuit design problem in such a way that the issues previously mentioned are taken into account.

First, we emphasize the importance of replication by allowing the use of only one device: the multiplexer (Mux) with 1-control line. Then, we restate our goal so that now we are interested in generating fully functional circuits in which the total number of multiplexers used is minimum.

The organization of this paper is the following: first, we will describe the problem that we wish to solve in a more detailed form. Then, we will introduce a methodology based on genetic programming to synthesize logic functions using multiplexers. Finally, we will show some examples taken from the technical literature and we will discuss some of our results. We compare the number of muxes found in the minimized circuits against $2^n - 1$ muxes required by the standard implementation.

2. Statement of the problem

The problem of interest to us consists of designing a circuit that performs a desired logic function (specified by a truth table), using the least possible number of 1-control line multiplexers. As will be described below, a logic function with n variables can be implemented using 2^n-1 1-control line multiplexers. Any implementation using less than that number of elements could be considered an improvement in the design. Since the optimal minimum number needed is unknown for most of the logic functions, the use of a heuristic such as genetic programming [7] seems adequate.

The implementation cost measured in terms of silicon surface has been studied for many years. Consider an n-variable multiplexer realized by means of $2^n - 1$ multiplexers with 1-control signal. Assuming that the cost of a single unit is K, the cost of such a realization is proportional to $K(2^n-1)$. Therefore, any realization with a fewer number of elements implies an improvement of the total manufacturing cost [4].

In our approach we permit only "1s" and "0s" to be fed into the multiplexers. Due to our goal, this marks a clear difference with other strategies where a variable can be fed into a mux. We allow the variables to be used only as the control signal of the multiplexer.

3. Previous work

It is possible to find in the literature several reports concerning the design of combinational logic circuits using GAs. Louis [11] was one of the first researchers who reported this class of work. Further work has been reported by Koza[1][7], Coello et al. [2, 3], Iba et al. [6], and Miller et al. [12].

However, none of these approaches has concentrated on the exclusive use of multiplexers to design combinational circuits using evolutionary techniques, although some researchers such as Miller [12] have used multiplexers as another permissible gate which can be combined with the traditional Boolean functions to design circuits.

Several strategies for the design of combinational circuits using multiplexers have been reported after the concept of *universal logic modules* [16]. Chart techniques [10], graphical methods for up to 6 variables [15], and other algorithms more suitable for programming have been proposed [13, 5, 1, 14]. The aim of these approaches (muxes in cascade or tree or a combination of both), is either to minimize the number of multiplexers, or to find p control variables such that a boolean function is realizable by a multiplexer with $p-$control signals.

4. Multiplexers as universal logic basis elements

A *binary multiplexer* with n *selection lines* is a combinational circuit that selects data from 2^n input lines and directs it to a single output line. The rationale supporting the use of this device as an *universal logic unit* is the following: any Boolean function of n variables can be implemented by a multiplexer with n-control signals. Furthermore, every n-control signals multiplexer can be synthesized by $2^n - 1$ 1-control signal multiplexers, as it is shown in Figure 1. Notice that the number of layers or depth of the array is equal to n.

Each unit copies either one of its inputs to the output depending on the state of the control signal. Multiplexers have two forms that we simply name *class A* and *class B*. For a *class A* multiplexer, when the control is set to one the input labeled as "1" is copied to the output, and vice-versa, the input labeled as "0" is copied to the output when the control is zero. For a *class B* multiplexer the logic is exactly the opposite: copy the input labeled "0" when the control line is one, and copy the input labeled "1" when the control is zero. In order to differentiate this property, class A muxes have the control signal on the right hand side and class B on the left, as can be seen in Figure 1. Therefore the control signal is located on the side of the input to be propagated when the control is in *active state* (The active state will be "1" for all our examples).

It is possible to use both classes of multiplexers simultaneously in a circuit. The design criteria could allow them as well. Two characteristic properties of circuits of this nature should be taken into consideration during the design process:

- **Class Transformation Property**: Class A and

[1] It is worth mentioning that Koza's approach to the design of combinational circuits has only concentrated on the generation of fully functional circuits and not in their optimization.

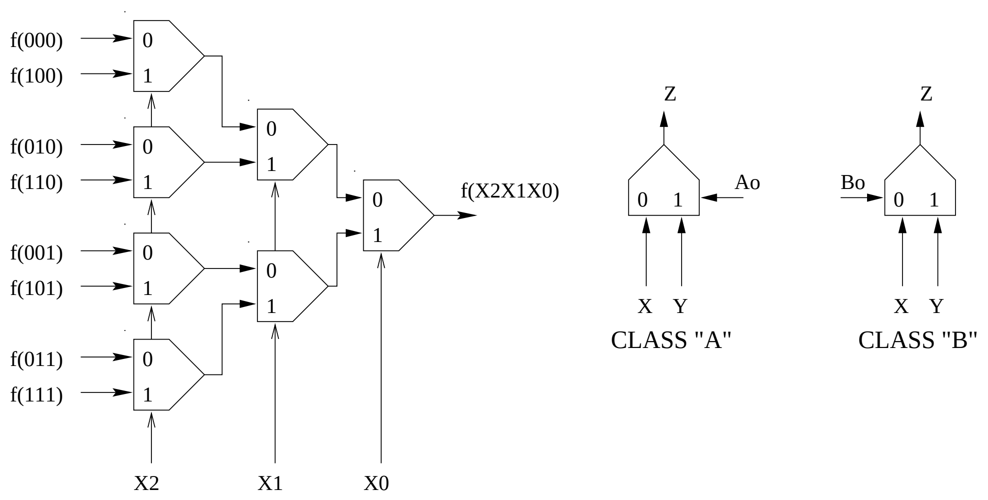

Figure 1. Implementation of a multiplexer of 3-control signals by means of 7 1-control signal muxes. Muxes class "A" and class "B". Functional equivalence between both classes

class B multiplexers can be converted freely from one class into the other, by just switching their inputs, thus input labeled "1" goes to input "0" and input labeled "0" now goes into "1" (see Figure 1).

- **Complement Function Property**: For every logic function F, its complement F' is derivable from the very same circuit that implements F by just negating the inputs, that is, by changing "0s" to "1s" and "1s" to "0s" (see Figure 2).

The correspondent consequences of these properties are the following:

- **Implementation in One Class**: Every circuit can be implemented by means of multiplexers of only one class (by the class transformation property). Since we are aiming to replicate the same element as many times as possible, this is a highly beneficial design quality, as can be appreciated in Figure 3.

- **The Minimum Circuit Equivalence**: If the function F and its complement F' are found to be implemented by particular and different size (i.e., number of elements) circuits, then both circuits are solutions for both functions (by the complement function property). Therefore, the smallest circuit is the desirable solution. This means that in practice the designer would have an alternate procedure to verify the quality of the solution.

5. Applying genetic programming

There are several issues of interest concerning the use of genetic programming in this domain, such as the encoding of candidate solutions, and the crossover and mutation operators. The representation chosen for our work is a binary tree representing a candidate circuit. Most of the elements of the population are functional circuits, therefore, they implement (at least partially) the expected logical behavior. Each node of the tree is a multiplexer whose children could be either another multiplexer or a leaf. The corresponding value of a leaf is, of course, zero or one. Since trees and lists are natural structures in *Prolog* we decided to use this programming language to implement our system so that our implementation could be considerably simpler to develop.

The issues taken into consideration for this application and the way in which they were approached are as follows:

- **Representation:** Binary trees encoding the population are represented by means of lists. Essentially each element of the list is the triplet $(mux, left-child, right-child)$ that encodes subtrees as nested lists. The tree captures the essence of the circuit topology allowing only the children to feed their parent node. In other words, the inputs of a multiplexer can only be chosen from the previous level, as shown in Figure 4.

  Both classes of binary multiplexers (as described in section 4) are implemented, then, for instance,

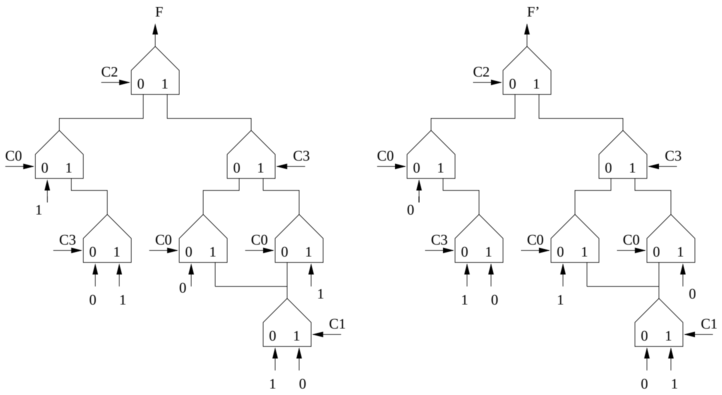

Figure 2. Transformation of a function into its complement function

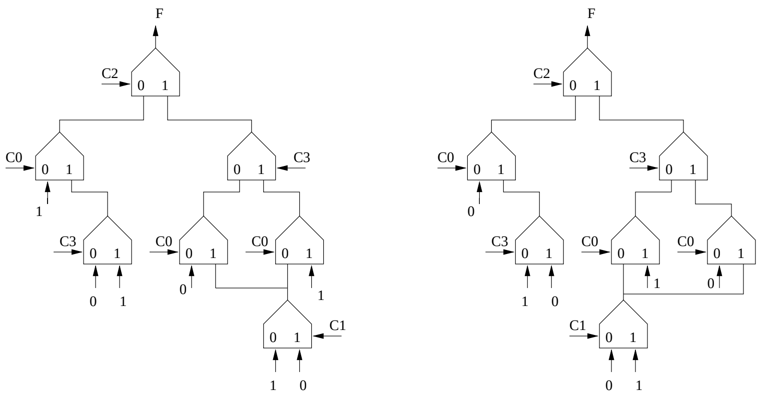

Figure 3. Transformation of a design into a circuit using only one class muxes ($Class\ B$ shown)

C2	C1	C0	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

(A2, (B1, (A0, 0, 1), 0), (B0, (B1, 0, 1), (B1, 1, 0)))

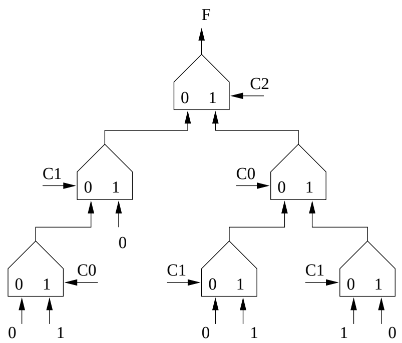

Figure 4. Truth table for logic function specification, circuit generated, and its coding

multiplexers $A0$ and $B0$ are activated by the same control bit $C0$ (the least significant bit in Figure 4) but they respond in a particular way to the activation state.

- **Crossover operator:** The exchange of genetic information between two trees is accomplished by exchanging subtrees. Our implementation does not impose any kind of restriction to the selection of subtrees or crossover points. Node-node, node-leaf, and leaf-leaf exchange are allowed. The particular case when the root node is selected to be exchanged with a leaf is disallowed, so that, no leaf may be mistakenly converted into a node thus avoiding the generation of invalid trees (in such cases the valid children are replicated twice).

- **Mutation operator:** Mutation is implemented in a simple way: first a mutation point is randomly chosen among the nodes and leaves. When a node (multiplexer) is selected, its control input is changed as follows (assuming n control signals): $a_0 \rightarrow a_1$, $a_1 \rightarrow a_2$, $a_{n-1} \rightarrow a_n$, $a_n \rightarrow a_0$. Similarly simple is the mutation of a leaf: $0 \rightarrow 1$, $1 \rightarrow 0$.

- **Fitness function:** Our goal is to produce a fully functional design (i.e., one that produces the expected behavior stated by its truth table) which minimizes the number of multiplexers used. Therefore, we decided to use a dynamic fitness function. At the beginning of the search, only compliance with the truth table is taken into account, and the evolutionary approach is basically exploring the search space. Once the first functional solution appears, we switch to a new fitness function in which fully functional circuits that use less multiplexers are rewarded. Regardless of the current stage of the fitness function, all members of the population have their fitness calculated in every generation. It is the fitness function the only agent responsible for the life span of the individuals.

- **Initial population:** The depth of the trees randomly created for the initial population is set to a maximum value equal to the number n of variables of the logic function (see Section 4). Considering complete binary trees, for n variables, $2^n - 1$ is the upper bound on the number of nodes allowed in the tree. However, we found in our experiments that in the initial population trees of shorter depth were created in larger numbers than trees of greater depth. This led us to adopt a strategy in which the size (depth and number of nodes) of the trees dynamically changed over generations. The goal was to allow our trees to grow without any particular boundaries as to allow a rich phenotypic variation in the population. Once the first fully functional circuit was found, then we switched to another fitness function that would minimize the total number of multiplexers used.

6. An informal algebra for terminal nodes

Our dynamic fitness function does not take into account the redundancy of the terminal nodes (nodes whose children are only 0 or 1), that is, identical terminal nodes are pruned away from any solution and counted as just another node. The function performed by a circuit clearly is a composition of node-functions over the children-node functions, and so on, all the way down to the bottom of the tree. With really low probability either one subtree might be identical to another subtree, or one subtree may consist of the same nodes. Therefore there are some redundant nodes that could be deleted by means of further analysis of the solution delivered by the genetic programming system. We have called it “redundancy and algebraic analysis”. Its goal is to improve in at least two ways the best solution found in the last generation. First, the number of elements might be reduced since redundant subtrees (if present) will be removed, and second, it opens the door to implementation decisions, for example, to select the preferred or recommendable class of multiplexer.

The only two rules that we used on the trees are shown in Figure 5.

Rule 1 is applied for transforming one multiplexer class into the other, aiming to maximize redundant nodes that can be deleted and the entire set replaced by just one of them. Subtrees as shown in rule 2 have been observed occasionally. By means of this rule we reduce the nodes of a subtree.

7. Comparison of results

We have used several circuits of different degrees of complexity to test our approach. For the purposes of this paper, 4 examples were chosen to illustrate the “evolved circuits” and the quality of the solutions produced using the metric previously mentioned. So far we have limited ourselves to the evolution of circuits with only one output.

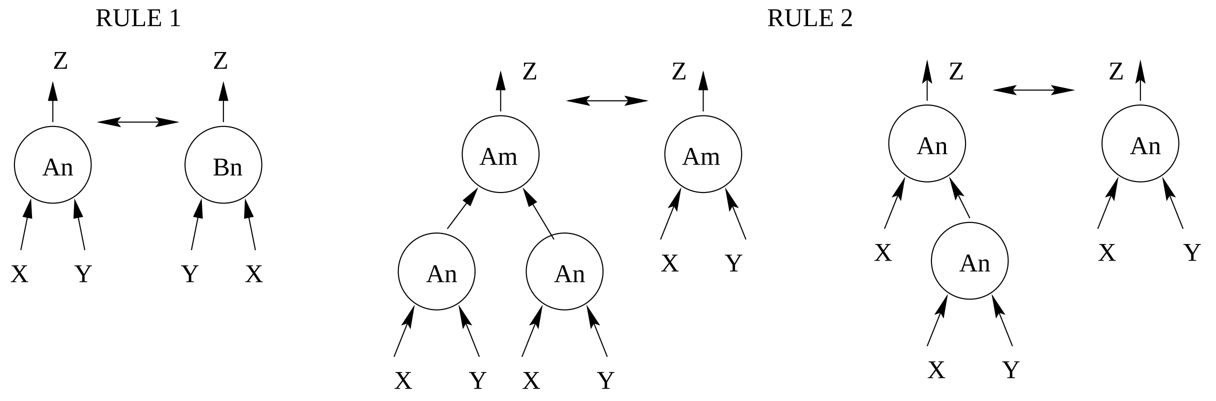

Figure 5. Redundancy and Algebraic Analysis. Node equivalence and subtree equivalence

7.1. Example 1

Our first example consists of a function with three inputs and one output, as shown in Table 1. It could be described in words as "the function whose output is 1 when only one of its inputs is 1. Otherwise the output is 0".
The solution has 5 multiplexers. Therefore, we have saved $(2^3-1)-5 = 2$ elements. This solution was found using an initial population of 600 elements evolved during 200 generations.

7.2. Example 2

In this second example we implement a function with four inputs and one output,as shown in Table 2. Only minterms are shown.(column F is shown for the sake of clarity)
The solution has 7 multiplexers. Therefore, we have saved $(2^4-1)-7 = 8$ elements. This solution was found using an initial population of 600 individuals evolved during 200 generations. The circuit found by the system and its simplified version after "redundancy and algebraic analysis" is shown in Figure 6

7.3. Example 3

In this third example we implement a function with 5 inputs and one output. The minterms of the function are the following:

$$f = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15, 18, 20, 21, 25, 26, 28,$$

$$30, 31)$$

The solution found has 15 multiplexers. Therefore, we have saved $(2^5-1)-15 = 16$ elements. This solution was found using an initial population of 600 individuals evolved during 1000 generations.

7.4. Example 4

In this last example we seek a solution for a more complex boolean function (6 variables). Next is the list of minterms of the function.

$$f = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15, 18, 20, 21, 25, 27, 28,$$

$$30, 31, 32, 33, 35, 38, 39, 40, 42, 45, 47, 50, 52, 53, 57, 59,$$

$$60, 62, 63)$$

The solution found has 21 multiplexers, Therefore, we have saved $(2^6 - 1) - 21 = 42$ elements. This solution was found using a population of 1500 individuals evolved during 700 generations.

Table 3 condenses the four examples. The column n-Mux shows the number 1-control line multiplexers needed to implement a n-control lines multiplexer. The column $GPOutput$ shows the number of nodes in the best solution found by our genetic programming system, and $GPAlgebraic$ is the further refined solution (the final circuit). $TotalSaved$ is the difference between columns 3 and 5.

8. Conclusions

We have shown a genetic programming approach for the synthesis of logic functions and minimization of their number of elements. We have been able to find solutions that require fewer elements than the standard array of $2^n - 1$ multiplexers of 1-control signal. Since the topology of the minimized solutions is pretty much that of a tree, we expect the VLSI implementation of this circuits to require very few extra work in routing algorithms to redesign the optimal connection paths or circuit layout.

C2	C1	C0	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

Table 1. Truth table for the first example.

C3	C2	C1	C0	F
0	0	0	0	1
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	1	0	1	1
1	1	1	1	1

Table 2. Minterms table for the second example.

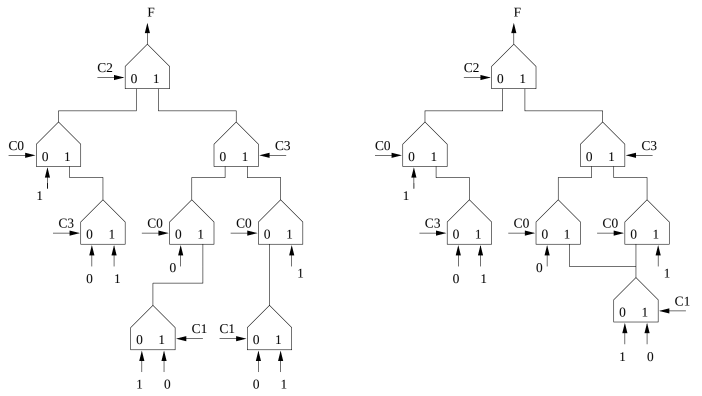

Figure 6. Circuit for the Example 2 and its simplified version after "redundancy and algebraic analysis"

Example	Inputs	n-Mux	GP Output	GP Algebraic	Total Saved
1	3	7	6	5	2
2	4	15	8	7	8
3	5	31	22	15	16
4	6	63	27	21	42

Table 3. Comparison on the size of the delivered circuits, optimized circuits, and standard implementation

9. Future work

We are interested in the improvement of our fitness function. We believe that a new fitness function that takes into account the number of repeated nodes could guide the search towards better solutions, reducing the amount of extra human work. We are concerned by the quick growth of some trees and the low increment of their fitness. In such cases a "tree prune" operation seems mandatory and some strategies need to be explored. Other design strategies will be implemented thus allowing the variables to be used both as inputs and as control signals of the multiplexer. Different size multiplexers will be allowed as well.

10. Acknowledgments

This paper describes research done in the Department of Electrical Engineering and Computer Science at Tulane University. Support for this work was provided in part by DoD EPSCoR and the Board of Regents of the State of Louisiana under grant F49620-98-1-0351.

The second author ackowledges support from CONACyT through project number I-29870 A.

References

[1] A. E. A. Almaini, J. Miller, and L. Xu. Automated Synthesis of Digital Multiplexer Networks. *IEE Proceedings Pt E*, 139(4):329–334, July 1992.

[2] C. A. Coello, A. D. Christiansen, and A. H. Aguirre. Using Genetic Algorithms to Design Combinational Logic Circuits. In C. H. Dagli, M. Akay, C. L. P. Chen, B. R. Farnández, and J. Ghosh, editors, *Intelligent Engineering Systems Through Artificial Neural Networks. Volume 6. Fuzzy Logic and Evolutionary Programming*, pages 391–396. ASME Press, St. Louis, Missouri, USA, nov 1996.

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

[4] M. Davio, J. P. Deschamps, and A. Thayse. *Digital systems, with algorithm implementation*. Wiley, New York, USA, 1983.

[5] R. Gorai and A. Pal. Automated synthesis of combinational circuits by cascade networks of multiplexers. *IEE Procedings Pt E*, 137(2):164–170, March 1990.

[6] H. Iba, M. Iwata, and T. Higuchi. Gate-Level Evolvable Hardware: Empirical Study and Application. In D. Dasgupta and Z. Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 260–275. Springer-Verlag, Berlin, 1997.

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

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

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

[10] G. G. Langdon. A decomposition chart technique to aid in realizations with multiplexers. *IEEE Transactions on Computers*, C–27(2):157–159, February 1978.

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

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

[13] A. Pal. An algorithm for optimal logic design using multiplexers. *IEEE Transactions on Computers*, C-35(8):755–757, August 1986.

[14] S. G. Shiva. *Introduction to Logic Design*. Scott, Foresman and Company, 1988.

[15] A. Tosser and D. Aoulad-Syad. Cascade networks of logic functions built in multiplexer units. *IEE Proceedings Pt E*, 127(2):64–68, March 1980.

[16] S. S. Yau and C. K. Tang. Universal Logic Modules and Their Application. *IEEE Transactions on Computers*, C–19(2):141–149, February 1970.