Gate-level Synthesis of Boolean Functions using Binary Multiplexers and Genetic Programming

Arturo Hernández-Aguirre
Bill P. Buckles
Department of Electrical Engineering and Computer Science
Tulane University, New Orleans, LA, 70118, USA
hernanda,buckles@eecs.tulane.edu

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

Abstract-

This paper presents a genetic programming based approach to the synthesis of logic functions by means of multiplexers. The approach uses the 1-control line multiplexer as the only design unit, designing any logic function (defined by a truth table) through the replication of this single unit. Our fitness function works in two stages: it finds feasible solutions, and then concentrates in the minimization of the circuits. The proposed approach does not take any knowledge from the application domain.

1 Introduction

The synthesis of logic circuits is a problem as old as the first computer. Initial steps in this direction were given by Shannon [17], who found important mathematical properties that led to the synthesis of Boolean functions. Akers [2] proposed binary decision diagrams as the vehicle to represent and minimize Boolean functions. Bryant [4] proposed ordered binary decision diagrams (OBDD). Both approaches are based in the manipulation of the nodes of the graph, thus, the initial graph is transformed into functional equivalent subgraphs. The repetitive application of two or three simplification rules (derived from the problem domain) have proven to be sufficient to minimize the graphs. In essence, the goal is achieved by a top-down minimization strategy, that is, reduced graphs are produced from complete graphs. Several researchers have added the decision diagram simplification rules into the genetic programming environment by modifying the crossover and mutation operators [20, 7]. The additioned knowledge reduces the convergence time, and improves the ability to find optimal solutions. Mechanisms of this sort do not synthesize circuits; they rather simplify a set of fully functional initial circuits (by reducing graphs). They also manifest lost of generality since the additioned knowledge makes its applicability specific. The genetic programming (GP) approach we are to describe follows the *automatic programming* capacity proclaimed by Koza [10], that is, GP synthesizes programs or functions that reproduce a desired behavior. In our system, GP constructs boolean functions by combining samples taken from the space of partial solutions. Once a 100% functional solution is found, our goal is turned to their minimization. Thus, the fitness function is updated to reward fully functional solutions with fewer elements. Trees (that represent circuits) are therefore trimmed, and nodes are replicated without having added any heuristic other than a simple change in the fitness function. The enormous difference in the approaches is evident. Graph techniques apply minimization rules to the binary decision diagrams. GP with additioned knowledge becomes a top-down reduction method [20, 7]. Our system works with the *purest* form of genetic programming, thus, no problem domain knowledge is included in the evolutionary process, and yet, it is able to sinthesize optimal or near optimal size circuits.

Our approach is analog to the"gate-level design". Commonly, gate-level design using evolutionary techniques make use of a sound and complete set of gates (AND, OR, NOT, XOR). Therefore, circuit synthesis is achieved by the correct composition (conexions) of a sound set of gates. [15, 9, 6, 12, 11]. We took a radical approach to the circuit design problem and we have substituted *gates* by *binary multiplexers*. Binary multiplexers are universal function generators (defined later), thus, they form a sound basis for the synthesis of logic functions. The working hypothesis is that GP can synthesize logic circuits by means of binary multiplexers [10], and that the replication of only one element (instead of five or six different gates) will decrease the manufacturing process cost (in this paper we address only the first issue). We emphasized the importance of replication by allowing the use of only 1-control line multiplexers in the evolutionary process. In our approach we permit only "1s" and "0s" to be fed into the multiplexers. Thus, we allow the variables to be only used as control signals of the muxes. In fact, this makes a clear difference to well known tabular strategies where a variable can be fed into a Mux (this restriction is also valid in OBDDs).

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. To end, we will compare optimal solutions found by other approaches (OBDD) with the solutions delivered by our GP system.

2 Problem Statement

The problem of interest to us is the design of a logic circuit that performs a desired Boolean function using the least possible number of 1-control line multiplexers (Mux). It is well

known that logic functions of n variables can easily be implemented by $2^n - 1$ 1-control line multiplexers. Likewise, what is widely unknown is the degree of redundancy of the solution. We address this problem in the first set of experiments. Further comparison is provided by contrasting our circuits against those created using OBDD techniques. In the last set of experiments we report an important circuit design problem: partially specified boolean functions.

3 Previous Work

It is possible to find in the literature several reports concerning the design of combinational logic circuits using GAs. Louis [14] was one of the first researchers who reported this class of work. Further work has been reported by Koza[1][10], Coello et al. [5, 6], Iba et al. [9], and Miller et al. [15]. However, none of these approaches has concentrated on the exclusive use of multiplexers to design combinational circuits using evolutionary techniques. Several strategies for the design of combinational circuits using multiplexers have been reported after the concept of *universal logic modules* [21]. Chart techniques [13], graphical methods for up to 6 variables [19], and other algorithms more suitable for programming have been proposed [16, 8, 3, 18]. 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. A popular approach named Ordered Binary Decision Diagrams (OBDD) make use of node transformations to reduce the size of the initial tree. Akers also shows a suitable transformation of trees into logic functions implemented by means of multiplexers [2]. Thus, multiplexers are only the implementation device (never seen during the design) while binary decision diagrams encode a Boolean function. Yanagiya [20] is credited as being the first to use OBDDs to learn the 20-multiplexer. After him, several researchers have included the OBDD minimization rules in the form of crossover and mutation operations into a GP based system (like Droste [7]). These systems perform circuit design by tree simplification and reduction.

4 Multiplexers as Function Generators

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 concept that supports the use of this device as an *universal logic unit* is known as *residues* of a function.

Definition 1. The residue of a boolean function $f(x_1, x_2, \ldots, x_n)$ with respect to a variable x_j is the value of the function for a specific value of x_j. It is denoted by f_{x_j}, for $x_j = 1$ and by $f_{\bar{x}_j}$ for $x_j = 0$.

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

Any Boolean function can then be expressed in terms of these residues in the form of an expansion known as Shannon's decomposition [17].

$$f = \bar{x}_j f|_{\bar{x}_j} + x_j f|_{x_j}$$

The logic function y that represent the mapping of two inputs A and B onto the output port of a multiplexer with one selector line s is: $y = sA + \bar{s}B$. This output function quickly takes the Shannon's expansion form if the same function is used in both input ports. Say $f = A = B$ is any logic function, then $y = sf + \bar{s}f$. If we pick x_j as the selector and the inputs are the residues f_{x_j} and $f_{\bar{x}_j}$, the output becomes $y = x_j \cdot f_{x_j} + \bar{x}_j \cdot f_{\bar{x}_j}$. Further expansion of the residues into selector-residue pairs leads to an expansion as shown in Figure 1. As can be observed, every n-control signals multiplexer can be synthesized by $2^n - 1$ 1-control signal multiplexers. Notice that the number of layers or depth of the array is equal to n.

Multiplexers can be "active low" or "active high" devices, a quality 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, or during the circuit synthesis. Two characteristic properties of circuits of this nature should be taken into consideration during the design process:

- **Class Transformation Property**: Class A and 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". Circuits can also be sinthesized using only one class of multiplexers.

5 The Genetic Programming Environment

In the following we describe genetic programming issues that should help to fully understand the approach. Representation, and the evolutionary operators: selection, crossover, and mutation are covered.

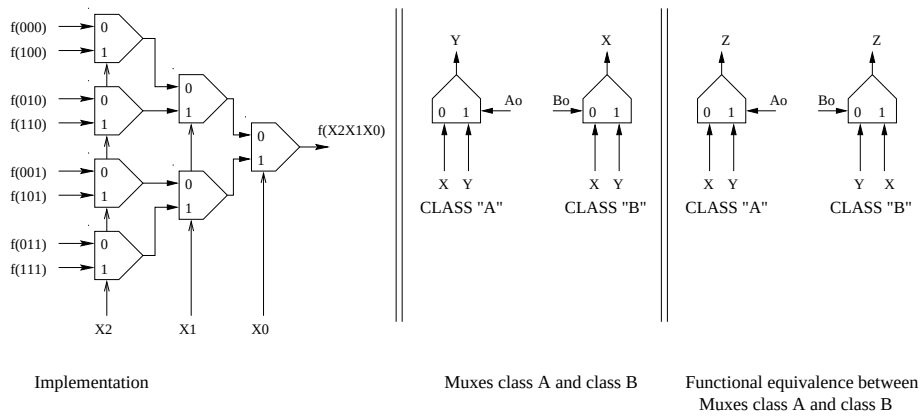

Figure 1: Implementation of a multiplexer of 3-control signals by means of 7 1-control signal muxes. Muxes class "A" and class "B". Functional equivalence between both classes

- **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 children to feed their parent node. In other words, a multiplexer takes only inputs from the previous level. This is shown in Figure 2.

  Both classes of binary multiplexers are implemented. Since multiplexers $A0$ and $B0$ are controlled by $C0$, the former is depicted with the control signal on its right side, and the latter with the signal on its left side.

- **Selection operator** The mating pool is created by ternary selection, thus, three individuals are randomly chosen from the entire population and the one with highest fitness is placed into the pool. The overall effect is the increment of the selection pressure that should decrease the convergence time.

- **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 two-stages 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 second fitness function in which fully functional circuits that use less multiplexers are rewarded. The fitness function is switched regardless of individuals that are not fully functional. The fitness function is 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. This is a fair limit because for complete binary trees with n variables, $2^n - 1$ is the upper bound on the number of nodes required. 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 allow the trees to grow without any particular boundaries as to allow a rich phenotypic variation in the population.

6 Further Refinement of the Solutions

Our two-stages fitness function does not take into account the redundancy of the terminal nodes. It simply rewards shorter trees with higher credit. Nonetheless, terminal nodes are usually replicated in vast numbers. Indirectly, this property provides for further minimization because duplicated terminal nodes are pruned away from the solution. Terminal nodes are deleted accordingly to the rules shown in Figure 3. Similar rules derived from the problem domain are given in [2]. Rule 1 is applied for transforming one multiplexer class into the other, aiming to maximize redundant nodes that can be

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

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

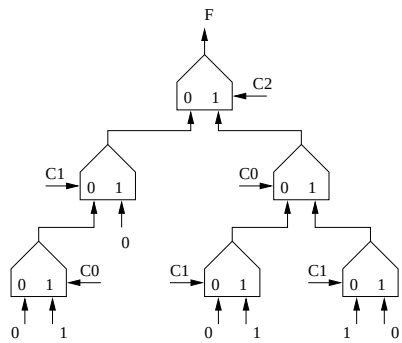

Figure 2: Truth table for logic function specification, circuit generated, and its coding

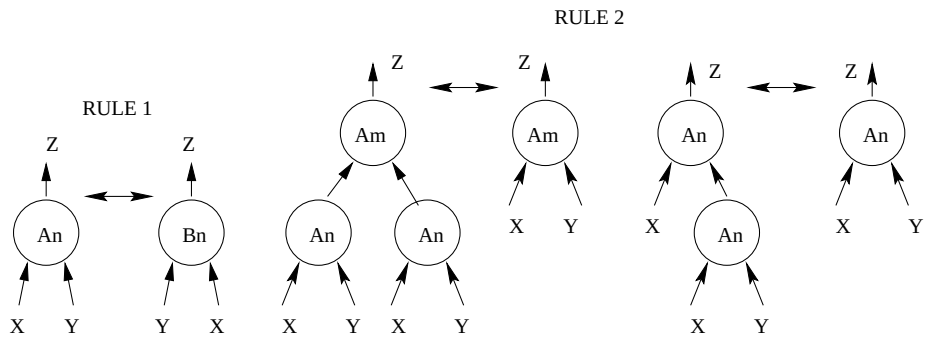

Figure 3: Further refinement. Node equivalence and subtree equivalence

deleted and the entire set replaced by just one of them. Subtrees as shown in rule 2 have been observed occasionally.

7 Experiments

The design metric in the following experiments is the number of elements. We contrast solutions against the standard implementation, ordered binary decision diagrams, and the design of partially specified boolean functions.

7.1 GP - Standard implementation

Using the standard implementations, Boolean functions with n variables can be implemented using $2^n - 1$ binary multiplexers. A considerable reduction in the number of elements is achieved by our system. We ran this experiments with a population size of 600 individuals. Maximum number of iterations is 100 for $F1$, 200 for $F2$, $F3$, and 700 for $F4$. This functions are found elsewhere.

Functions implemented

- $F1(a, b, c) = \sum(1, 2, 4)$
- $F2(a, b, c, d) = \sum(0, 4, 5, 6, 7, 8, 9, 10, 13, 15)$
- $F3(a, b, c, d, e) = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15, 18, 20, 21, 25, 26, 28, 30, 31)$

F	Vars	SI	GP	Saved
1	3	7	5	2
2	4	15	7	8
3	5	31	15	16
4	6	63	21	42

Table 1: Comparison GP-standard implementation

- $F4(a, b, c, d, e, f) = \sum(0, 1, 3, 6, 7, 8, 10, 13, 15$ $18, 20, 21, 25, 27, 28, 30, 31, 32, 33, 35, 38, 39, 42,$ $42, 45, 47, 50, 52, 53, 57, 59, 60, 62, 63)$

The table 1 condenses the results of these experiments. First column is the function implemented (itemized above), next the number of variables of that function, then the number of muxes required by the standard implementation, then the number of muxes required by our GP system. The last column shows the savings in the number of muxes, thus the difference between SI and GP.

7.2 GP - OBDDs

In the second set of experiments we contrast the evolved solutions against solutions delivered by OBDDs. It is widely known that OBDD are very sensitive to the node order, thus, circuit design in that case is mostly reduced to the computation of the variable order that minimizes the size of the circuit.

Since our interest is in the reduction of the number of nodes, some circuits are found to have less elements than in their optimal OBDD version.

Functions implemented

- $F5(a,b,c,d,e,f) = ab + cd + ef$
- $F6(a,b,c,d,e,f) = ad + be + ef$
- $F7(a,b,c) = a \oplus b \oplus c$ "odd-parity"

Function F5. The OBDD of any function similar to $F5$ with n variables has n nodes. The optimal order of the variables is $1, 2, 3, 4, 5, 6, \ldots, n$. [4]. We have found optimal solutions to functions of this sort with 4, 6, 8 and 10 variables. In Figure 4 the OBDD tree is depicted along with evolved solution. The subtree $b5$ is repeated on both branches. Even more, $b5$ can be further minimized. Careful count indicates that the evolved tree is optimal.

The genetic programming system found the optimal solution at generation 300, population size=990 individuals, probability of crossover=0.35, and probability of mutation per individual=0.65 Therefore, probability of mutation per gene=0.65/L, where L is the total number of terminals plus non-terminals in the tree.

Function F6. The next design is the synthesis of a similar function with 6 variables. The optimal solution found by OBDD to this problem has 14 non-terminal nodes with variable ordering $1, 2, 3, 4, 5, 6$. Thus, no other variable ordering will find a better solution using OBDD techniques [4]. In Figure 5 we show the evolved optimal solution delivered by the genetic programming system. It is implemented with only 10 nodes.

The genetic programming system found the optimal solution at generation 219, population size=990 individuals, probability of crossover=0.35, and probability of mutation per individual=0.65 Therefore, probability of mutation per gene=0.65/L, where L is the total number of terminals plus non-terminals in the tree.

Function F7. The "odd parity" function is a very hard problem to solve using multiplexers and genetic programming. In fact we have only found optimal solutions for up to 4 variables. Its hardness is due in part to the fact that there exist an ideal solution using xor gates. Therefore, any other approach will have more elements that the number of xor. Using OBDD, the solution for n variables has at most $2n - 1$ non-terminal nodes. In Figure 6 we show the OBDD solution, and the evolved optimal solution delivered by the genetic programming system with 7 nodes that can be reduced to 5.

The genetic programming system found the optimal solution at generation 26, population size=510 individuals, probability of crossover=0.35, and probability of mutation per individual=0.65 Therefore, probability of mutation per gene=0.65/L, where L is the total number of terminals plus non-terminals in the tree.

Input	F
0000	0
0001	1
0010	1
0100	1
1000	1
0111	1
1011	1
1101	1
1110	1
1111	0

Table 2: Partial function of 4 variables

k	vars	size	aver
2	4	7	60
3	8	15	200
4	16	31	700

Table 3: Convergence to the optimum

7.3 Partially specified functions

We want to address the ability of the system to synthesize circuits of optimal size for boolean functions with "large" number of variables. The following property allow us to verify our GP system. Boolean functions with 2^k variables, where ($k = 1, 2, \ldots$), is implemented with exactly $(2 \cdot 2^k) - 1$ binary muxes. For example, for $k = 2$, a boolean function of $2^2 = 4$ variables is implemented with exactly 7 muxes when the truth table is specified as shown in table 2. A similar technique has been used by Droste to specify the 11-multiplexer [7].For greater k, that is, number of variables, we specify the table in a similar way. There are exactly $2 \cdot 2^k + 2$ entries in the table.

The table 3 shows the high rate of convergence of the GP system to the optimum. We ran 100 experiments for each function (each k). Column "vars" is the number of variables for some integer k, "size" is the optimum number binary muxes needed to implement the partial boolean function, and "aver" is the average number of iterations needed to find the optimum. In all cases we found optimum size circuits in more than 90% of the iterations.

8 Final remarks

We have shown a genetic programming approach for the synthesis of logic functions and minimization of their number of elements. The systems delivers quite smaller circuits than the standard implementation does. Solutions delivered by our GP system are quite similar in size to the solutions delivered by OBDDs. More experimentation is needed in this respect. The ability to generate large partial functions has been verified to be optimal (in most cases) for functions with up to 16 variables. Some specific problems, as the "odd-parity" turned

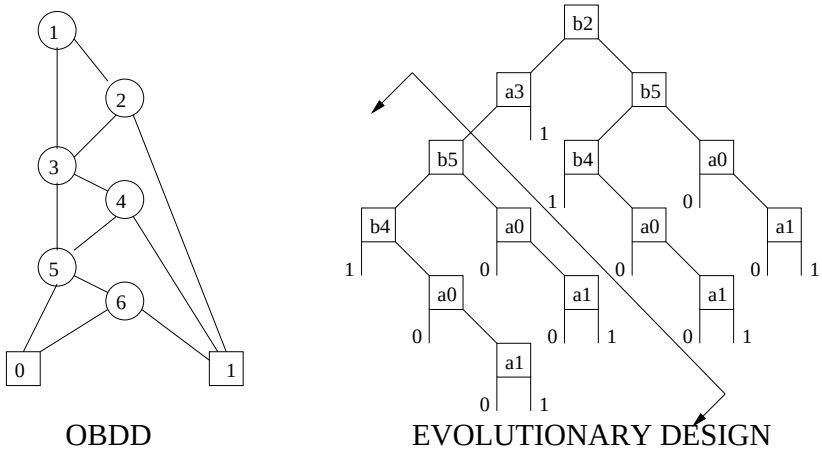

Figure 4: Synthesis of problem design 1

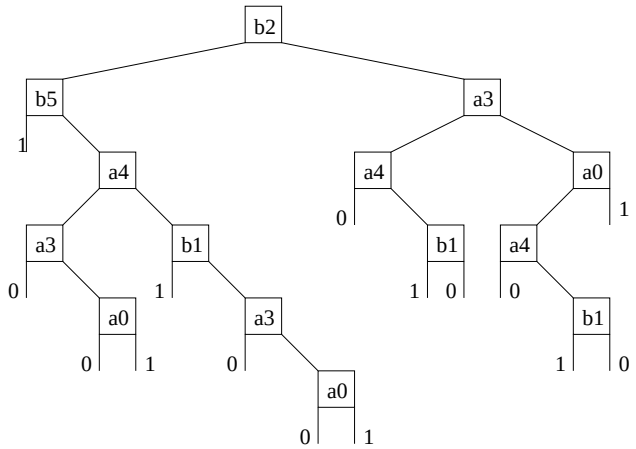

Figure 5: Synthesis of problem design 2

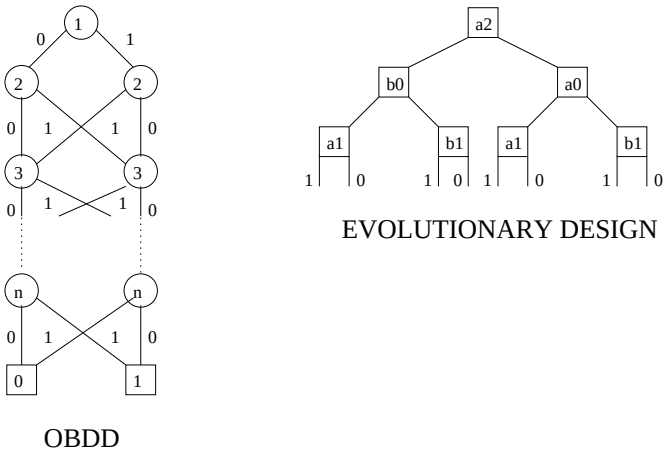

Figure 6: Synthesis of problem design 3

to be very hard to solve. We know the optimal solutions are not achievable by multiplexers but by the composition of xor gates.

9 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 third author acknowledges support from CONACyT through project number I-29870 A.

Bibliography

[1] A. Hernández Aguirre, C. Coello Coello, and B. P. Buckles. A genetic programming approach to logic function synthesis by means of multiplexers. In D. Keymeulen A. Stoica and J. Lohn, editors, *The First NASA/DoD Workshop on Evolvable Hardware*, pages 46–53. IEEE Computer Society, 1999.

[2] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[3] A. E.A. Almaini, J.F. Miller, and L. Xu. Automated synthesis of digital multiplexer networks. *IEE Proceedings Pt E*, 139(4):329–334, July 1992.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[5] Carlos A. Coello, Alan D. Christiansen, and Arturo Hernández Aguirre. Using genetic algorithms to design combinational logic circuits. In Cihan H. Dagli, Metin Akay, C. L. Philip Chen, Benito R. Farnández, and Joydeep 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.

[6] Carlos A. Coello, Alan D. Christiansen, and Arturo Hernández 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.

[7] Stefan Droste. Efficient genetic programming for finding good generalizing boolean functios. In John R. Koza, K. Deb, M. Dorigo, and D. Fogel, editors, *Proceedings of the Second Anual Conference on Genetic Programming*, pages 82–87, San Francisco, California, July 1997. Morgan Kaufmann.

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

[9] Hitoshi Iba, Masaya Iwata, and Tetsuya Higuchi. Gate-Level Evolvable Hardware: Empirical Study and Application. In Dipankar Dasgupta and Zbigniew Michalewicz, editors, *Evolutionary Algorithms in Engineering Applications*, pages 260–275. Springer-Verlag, Berlin, 1997.

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

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

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

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

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

[15] 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.

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

[17] C. E. Shannon. The synthesis of two-terminal switching circuits. *Bell System Technical Journal*, 28(1), 1949.

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

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

[20] Masayuki Yanagiya. Efficient genetic programming based on binary decision disgrams. In Toshio Fukuda and Takeshi Furuhashi, editors, *Proceedings of the 1995 IEEE International Conference on Evolutionary Computation*, pages 234–239, Nagoya, Japan, 1995. IEEE, IEEE.

[21] Stephen S. Yau and Calvin K. Tang. Universal logic modules and their application. *IEEE Transactions on Computers*, C-19(2):141–149, February 1970.