Circuit Design Using Genetic Programming: An Illustrative Study

Arturo Hernández-Aguirre
Centro de Investigación en Matematicas (CIMAT)
A. P. 402, Guanajuato, Gto.
C.P. 36000, MEXICO

Bill P. Buckles
Dept. of Elect. and Computer Science
Tulane University
New Orleans, LA 70118
USA

Carlos Coello-Coello
CINVESTAV-IPN
Depto. de Ingeniería Eléctrica
Sección de Computación
Av. Instituto Politécnico Nacional No. 2508
Col. San Pedro Zacatenco
México, D. F. 07300 MEXICO

Abstract – **We present here a genetic programming based approach to the synthesis of logic functions via multiplexers. This approach, a bottom-up synthesis procedure, closely follows the "automatic programming" goal of GP. Our method uses the 1-control line multiplexer as the only design unit for the synthesis of any logic function. Logic design with multiplexers is similar to logic design with Binary Decision Diagrams, which can be transformed into Ordered Binary Decision Diagrams. Since the metric of our designs is the minumum number of components, ordered diagrams are not a suitable approach. A larger issue that we wish to argue, however, is that GP, given the right solution representation, is a competitive approach to circuit design.**

1 Introduction

The minimization of logic circuits is a problem born with the first computer. The evolutionary computation approach we describe follows an approach opposite of the manipulaton of binary decision trees [2, 5]. Such approaches are essentially top-down and involve the repetitive application of node rules such as remove terminals, remove nonterminals, remove redundant tests [5], and *reduce* [10].

A bottom-up searching procedure (genetic programming) constructs boolean functions by combining samples taken from the space of partial solutions. Both find a functional solution having the fewest elements. The difference in the approaches is evident: In graph techniques the "minimization rules" are derived from the problem domain. In our evolutionary system we work with the *purest* form of genetic programming, thus, no problem domain knowledge is included.

The gate-level design of combinational circuits by means of evolutionary computation has been approached only recently [17, 11, 7, 14, 13]. We propose an approach for which we encode a population (candidate solutions) of circuits using trees [1]. The trees encode Boolean functions for which every non-terminal node is a binary multiplexer. The hypothesis is that the replication of simple and elemental binary multiplexers is a sound process for the synthesis of logic functions. We emphasized the importance of replication by allowing the use of only 1-control line multiplexers in the evolutionary process.

4.1.2

There exists an isomorphism between trees representing Boolean functions and the multiplexer-based implementation. Binary trees can be transformed into binary decision diagrams (BDD), BDDs tranformed into Ordered Binary Decision Diagrams (OBDD), and OBDDs into Reduced OBDDs (ROBDDs). ROBDDs provide a canonical representation of logic functions (the diagram representing the function is unique), therefore, they are a powerful tool for reasoning about logic functions. For instance, to determine whether two Boolean functions are identical, one may verify whether their ROBDDs are isomorphic.

2 Problem Statement

The problem of interest consists of designing a circuit that performs a desired logic function using the least number of 1-control line multiplexers (Mux). Any logic function with n variables can be implemented using $2^n - 1$ 1-control line multiplexers. In our approach we permit only "1s" and "0s" as input to the multiplexers. We allow the variables only as control signals. This is a clear difference to well known tabular strategies in which a variable can be fed into a Mux.

It is possible to find in the literature reports of the design of combinational logic circuits with GAs. Louis [16] was one of the first researchers who reported this class of work. Further work has been reported by Koza [12], Coello et al. [6, 7], Iba et al. [11], and Miller et al. [17]. None of these approaches has concentrated on the exclusive use of multiplexers. The strategies for the design of combinational circuits using multiplexers have been are related to the concept of *universal logic modules* [22]. Chart techniques [15], graphical methods for up to six variables [21], and other algorithms more suitable for programming have been proposed [18, 9, 3, 20]. 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.

The popular approach named Ordered Binary Decision Diagrams (OBDD), make use of node transformations to reduce the size of the initial tree. Trees of Boolean functions with n variables have size $2^n - 1$, that is, the number of non-terminal nodes. It is also shown in [2] the suitable transformation of trees into logic functions by means of multiplexers. Thus, we will compare the circuits delivered by BDD against the circuits delivered by our evolutionary system.

3 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 $x_j = 0$.*

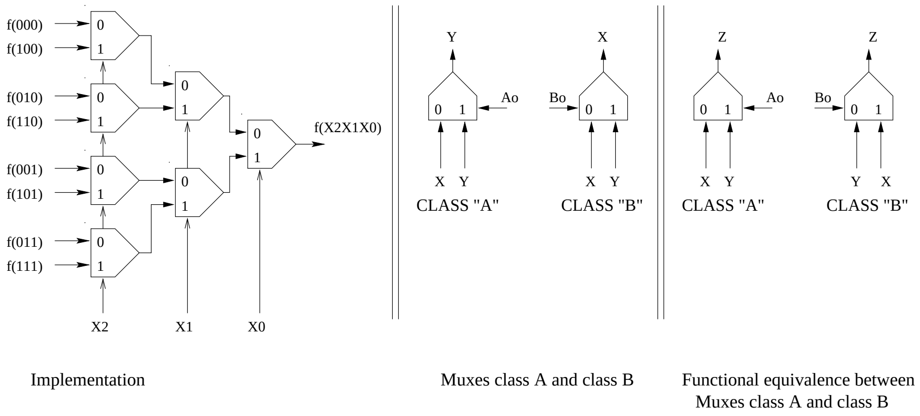

Figure 1: Implementation of a multiplexer of 3-control signals by means of 7 1-control signal muxes. Muxes class "A" and class "B". Functional equivalence between both classes

Any Boolean function can then be expressed in terms of residues in the form of an expansion known as Shannon's decomposition [19].

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

The logic function y that maps two inputs A and B onto the output port of a multiplexer having one selector line s is: $y = sA + \bar{s}B$. This output function takes the Shannon's expansion form if the same function is used at 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 is 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 side and class B on the left, as show 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 our examples.)

4 The Genetic Programming Environment

We describe below the genetic programming issues that should assist in understanding the approach – representation, and the evolutionary operators (selection, crossover, and mutation).

4.1.4

(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

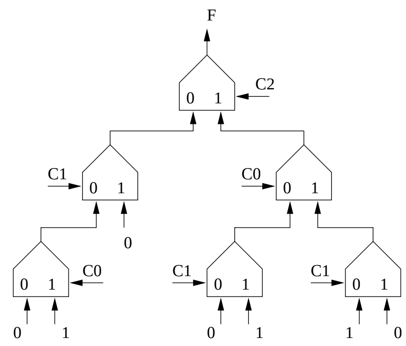

Figure 2: Truth table for logic function specification, circuit generated, and its coding

- **Representation** Binary trees encoding the population are represented by means of lists. Essentially each element of the list is the triplet $(mux, left-child, right-child)$ and subtrees are nested lists. 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 population and the one with highest fitness is placed into the pool. Thus, selection pressure is high.

- **Crossover operator** The exchange of genetic information between two through exchanging subtrees. Our implementation does not impose a restriction on the selection of crossover points. Node-node, node-leaf, and leaf-leaf exchange are allowed. Not allowed is the selection of the root node to be exchanged with a leaf.

- **Mutation operator** Mutation is implemented in a simply: First a mutation point is randomly chosen. When a node (multiplexer) is selected, its control input is changed as follows (assuming n control signals): $a_0 \rightarrow a_1$, $a_1 \rightarrow a_2$, $a_{n-1} \rightarrow a_n$, $a_n \rightarrow a_0$. The mutation of a leaf is: $0 \rightarrow 1$, $1 \rightarrow 0$.

- **Fitness function** Our goal is to produce a fully functional design (i.e., one that produces the expected behavior stated by its truth table) and minimizes the number of multiplexers. Therefore, we use a two-stages fitness function. At the beginning of the search, only compliance with the truth table is taken into account. The evolutionary search is basically exploritory. Once the first functional solution appears, we switch to a fitness function in which fully functional circuits that use fewer multiplexers are rewarded. All members of the population have their fitness calculated in every generation.

- **Initial population** The depth of the trees randomly created for the initial population is set to a maximum equal to the number n of variables in the logic function. For complete binary trees with n variables, $2^n - 1$ is the upper bound on the number of nodes required. Thus, a depth of n is reasonable. However, we found that in the initial population trees having a shorter depth were created more frequently than trees of greater depth. Due to this, we allow the trees to grow without particular boundaries to allow a rich phenotypic variation in the population.

5 Experiments

We contrasted solutions with the standard multiplexer implementation [1]. A considerable reduction in the number of elements was achieved. Specifically, we contrast evolved solutions against solutions delivered by OBDD. It is known that OBDD are sensitive to node order, thus, circuit design in this case is mostly reduced to the computation of the variable order that minimizes the circuit. Our design metric is the *number of nodes*, therefore, we expect to find the same number of nodes as for the optimal case delivered by OBDD, regardless of the variable order (when the optimal case is known).

5.1 Problem design 1

The first boolean function we want to synthesize has 6 variables: $F = X_1X_2 + X_3X_4 + X_5X_6$. The OBDD of any function of this sort 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 with 4, 6, 8 and 10 variables. In Figure 3, the ROBDD tree is depicted along with evolved solution. Both circuits have the same number of nodes although the variable ordering is different. For the evolved solution, variable ordering is not a defining parameter, but the correct application of this result is that the variable ordering {3,4,6,5,1,2} is also optimal for a ROBDD implementing $F = X_1X_2 + X_3X_4 + X_5X_6$.

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.

5.2 Problem design 2

The next design is the synthesis of another function with 6 variables: $F = X_1X_4 + X_2X_5 + X_3X_6$. The optimal solution found by ROBDD to this problem has 14 non-terminal nodes with variable ordering $1, 2, 3, 4, 5, 6$ (see [4] for a discussion on the encoding of this function). In Figure 4, we show the ROBDD solution (from [4]), and the evolved optimal solution delivered by the genetic programming system. It is implemented with only 10 nodes. The comparison The evolved solution does not include the ordering restriction. The point we make is that less specialized tools are more crafty.

The GP system found the optimal solution at generation 219, population size=990, probability of crossover=0.35, and probability of mutation per individual=0.65. Therefore, prob-

4.1.6

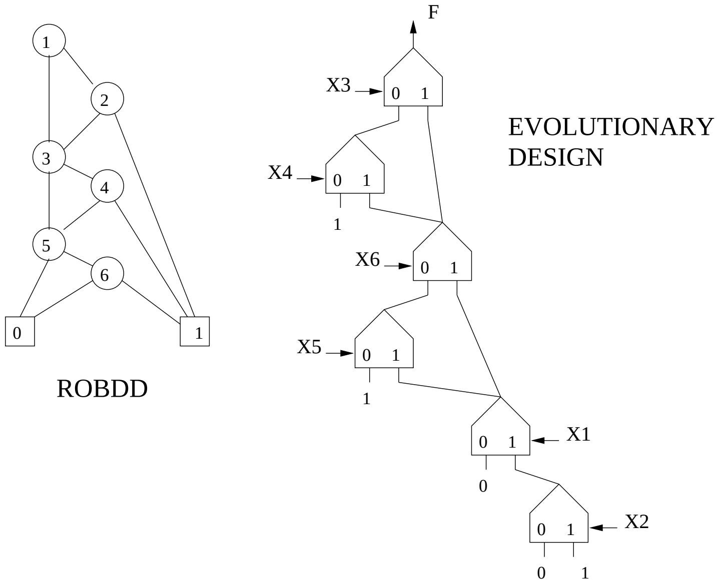

Figure 3: Synthesis of problem design 1, $F = X_1X_2 + X_3X_4 + X_5X_6$

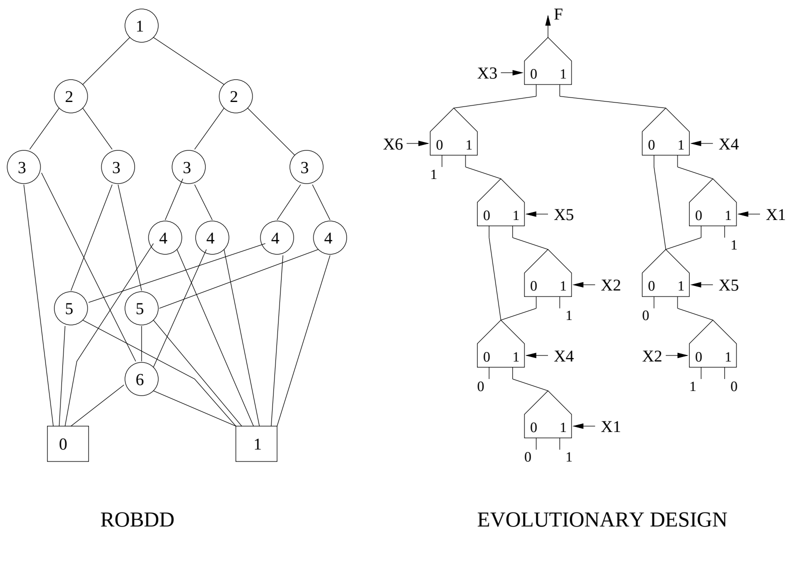

Figure 4: Synthesis of problem design 2, $F = X_1X_4 + X_2X_5 + X_3X_6$

ability of mutation per gene=0.65/L, where L is the total number of terminals plus non-terminals in the tree.

5.3 Problem design 3

The "odd parity" function is a hard problem to solve using multiplexers and GP. In fact we found optimal solutions for up to 4 variables. The Boolean function of 3 variables is: $F = X_1 \oplus X_2 \oplus X_3$. Using OBDD, the solution for n variables has at most $2n-1$ non-terminal nodes. In Figure 5 we show the OBDD solution, and the evolved optimal solution delivered by the genetic programming system with seven nodes that can be reduced to five.

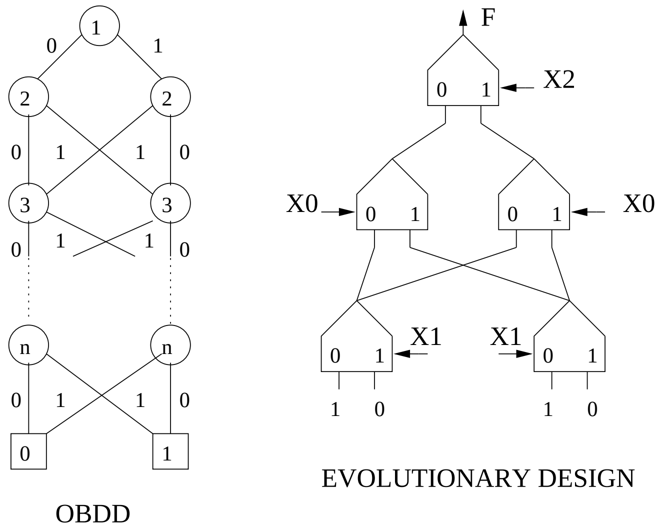

Figure 5: Synthesis of problem design 3, $F = X_1 \oplus X_2 \oplus X_3$

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.

5.4 Problem design 4

The following problem is inspired by what is known as the *11-multiplexer* and *20-multiplexer* problems. Droste [8] has shown a partially specified function approach to these problems. We wish to verify the ability of the system for designing Boolean functions with a "large" number of arguments and specific topology. That is, the topology is preserved as the number of variables increases. Boolean functions with 2^k variables (where $k = 1, 2, \ldots$), are 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 1. For any k (i.e., the number of variables), we specify the table in a similar way. Notice that there are exactly $2 \cdot 2^k + 2$ entries in the table.

Table 2 shows the high rate of convergence of the GP system to the optimum. We ran 100 experiments for each function (each k). The column called **vars** shows the number of variables for some integer k, **size** refers to the optimum number of binary muxes needed to

4.1.8

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

Table 1: Problem design 4, partially specified function of 4 variables

implement the partial Boolean function, and **aver** indicates 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.

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

Table 2: Convergence to the optimum

6 Conclusions

We have shown a genetic programming approach for the synthesis and minimization of logic functions. We have seen shown that the delivered solutions agree with the known optimal cases of OBDD for the optimal variable ordering, and in some other cases the evolved solution require less number of elements. The driving forces of evolutionary computation techniques are, at least, as efficient as the minimization rules derived from particular problem domains, such as OBDD.

7 Acknowledgments

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 partial support from REDII-CONACyT, and from CONACyT project number I-29870 A.

References

[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] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computer Surveys*, 24(3):293 – 318, September 1992.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambrige University Press, 2000.

[11] 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.

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

[13] 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.

[14] 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.

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

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

[17] 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.

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

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

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

[21] 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.

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