Decreasing Transistor Count Using an Edges Sharing Technique in a Graph Structure

Vinícius N. Possani, Luciano V. Agostini, Felipe S. Marques, Leomar S. da Rosa Jr.
{vnpossani, agostini, felipem, leomarjr}@inf.ufpel.edu.br

Group of Architectures and Integrated Circuits – GACI
Federal University of Pelotas – UFPel
Pelotas – Brazil

Abstract

Increasingly, in VLSI designs, the integrated circuits have higher density of transistors on the small physical area, power consumption reduced and greater performance. An important factor that has contributed for this is the representation of logic functions with a reduced number of transistors. Thus, we sought an alternative solution to common methods, such as factorization, to generate optimized networks. This paper presents a graph-based structure to represent a transistor network and a technique to reduce the number of transistors by edges sharing. Our method can achieve non-series-parallel arrangements while methods based in factorization can only derive series-parallel arrangements, which may not be the best solution. Thus, when applied to the set of 4 input p-class logic functions, our method has advantages if compared to the good-factor algorithm implemented in SIS Software. Also, in other logical functions our algorithm can achieve results as good as those generated by techniques based in BDD.

1. Introduction

The micro electronics industry has brought great advances in last years, no doubt, designing digital circuits VLSI becomes an increasingly task of extreme complexity and high cost of resources and time. In this context, aid tools are applied to support these projects, contributing to the designers manipulate more transistors and decreasing the development cycle. Therefore, the automatically generation of transistor networks makes simple some arduous tasks. Moreover, it also reduces the aggregate cost to the final product.

This paper proposes an edge sharing method, on a graph structure, to generate optimized transistor networks. In our approach, the input Boolean expression is translated into a graph that is later optimized through edges sharing. Nowadays, alternative methods which are available in the literature has been study and applied in this context. They are based on graph optimizations, were each edge in the graph keeps an association with a transistor in the network. The main idea is try to minimize the edges in an existent graph [1] or to compose a new graph with a reduced number of edges [2]. These alternative methods are used because the common technique to optimize a transistor network is based on factorization [3-4] and this may not be an optimum solution [5]. In factorization method an input Boolean expression is manipulated in order to reduce the number of literals that compose the expression. Subsequently, this optimized expression is translated in a transistor network composed by a reduced number of switches. In this sense, our sharing method intent derives non-series-parallel arrangements in order to deliver better results than the common technique.

2. Edges Sharing Method

The edges sharing method considers as input a sum-of-products (SOP) expression. In order to translate the expression to a graph, a parser is needed. The parser will deliver one vector of literals for each product storing these vectors in a list. Afterward, it is started the assembly of the graph by removing vectors one at a time from this list and creating an edge in the graph for each literal found in the vector. As an example we will use the Exp. (1) which represents a 'XOR' with 4 inputs. Fig. 1.a shows the graph obtained of this expression.

!A*!B*!C*D + !A*!B*C*!D + !A*B*!C*!D + !A*B*C*D +
A*!B*!C*!D + A*!B*C*D + A*B*!C*D + A*B*C*!D (Exp.1)

In the sequence, all paths in the graph are traversed in order to recognize identical edges (edges that represent same literals and have at least one vertex in common). If this condition is verified in the graph, then the identical edges are shared. This procedure consists in keeping only one of these identical edges, eliminating the remaining edges and merge the vertices that connect them. The vertices that will be merged are detached with the circumferences without fill in the figures below. This is exemplified in Fig. 1.b where the edge '!A' was shared and the vertices 1, 5, 8 and 11 were merged. Now the edge 'A' will be shared generating the graph shown in Fig. 1.c. So, in this moment the vertices 8 and 17, one at a time, are considered the new starting point of the optimization process, where the algorithm sought identical edges between these two vertices and the vertex 4. This way the edge '!B' connected to the vertex 8 will be shared and in sequence this occurs with edge 'B'. Afterward that, the same process is applied to the edges '!B' and 'B' attached to the vertex 17. This is demonstrated in Fig. 1.d.

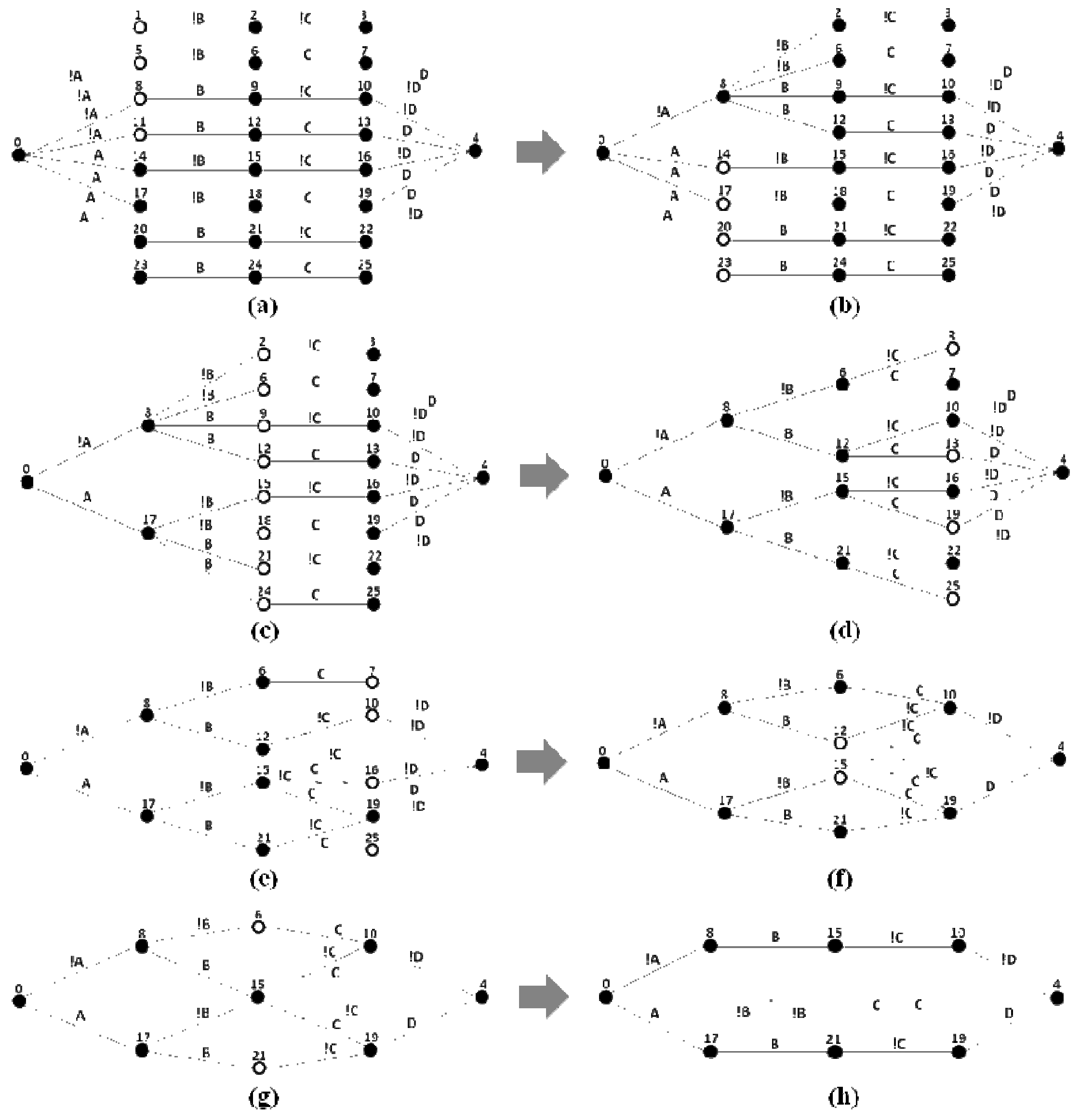

Fig. 1 – Steps of the sharing method to a 'XOR' with four inputs.

Considering the Fig. 1.d, departing from the vertices 6, 12, 15 and 21, one at a time, and traversing the graph toward the vertex 4, it is not possible to perform new optimizations because identical edges are not found between these vertices and the vertex 4. To perform new optimizations the sharing algorithm is applied from the end to the beginning of the graph. Thus, departing from the vertex 4 the edges 'D' are identified and were shared as the Fig. 1.e. demonstrates. In the next step the edges '!D' will be shared resulting in the graph of the Fig. 1.f.

Now, consider the vertices 10 and 19 as new start points of the sharing algorithm. There are two edges '!C' connected on the vertex 10, as Fig. 1.f shows. Then it is possible to remove the edge '!C' attached to the vertices 10 and 12 merging the vertices 12 and 15. In this case, the merging of the vertices 12 and 15 will derive two edges 'C' between the vertices 19 and 15. When this is detected, just one of these edges remains in the graph. Fig. 1.g shows this state of the graph. This process is applied again, but this time to the edges 'C' that are connected to the vertex 10, merging the vertices 6 and 21. This will derives two edges '!C' between the vertices 19 and 21, one of this edges will be removed resulting in the final graph illustrated by Fig. 1.h. Afterward, starting from the vertex 19, it is not possible to perform other optimizations. Thus, the optimization process ends. If any of these processes generates an invalid path, a recovery routine is invoked and the process is reversed. In the next session this procedure will be explained.

3. Validation Procedure

To guarantee that the optimized transistor network will be the faithful representation of the original expression, making sure that all products described in SOP are present in the resultant graph and sure that sneak-paths (forbidden paths) are not introduced on the network, a validation procedure is applied. Thus, it is necessary to validate all paths of the graph each time an edge is shared. The paths of the graph are generated through a recursive algorithm applied on the adjacency matrix that represents the graph. The algorithm uses a list structure to store the indexes of the matrix that make up a path of the graph.

Consider as example the simple graph illustrated by Fig. 2.a. The first step is to insert in the list the index '0' that indicates the row of matrix that represents the initial vertex of the graph. Then this row is traversed in searching of the literals. When the literal 'A' is found in the row '0' and column '1', as shown in Fig. 2.b, the index of this column is inserted into the list, if this index has not yet been inserted. Then, the algorithm immediately switches to the line '1' indicated by the column of the element found. Each row changing is a recursive call of the algorithm as Fig. 2.b demonstrates. This process is repeated recursively until the index '2' of the final vertex is reached, meaning that a path was formed. This procedure can be seen through the arrows in Fig. 2.b. Notice that literals in a column whose index has already been inserted in the list are ignored. Afterward, through the contents stored in the list it is possible to compose the path by indexing the matrix. All paths formed are stored in a list in order to compare with the original expression.

Once a path is formed, the last index '2', inserted into the list, is removed and the algorithm returns to the last cell of the matrix that was visited. Then, it continues traversing this row until finding another element, in this case the literal 'C', and it switches to the row '3' indicated by the column of this element.. If the row ends and any other element is found the algorithm returns to the last row visited and continues searching elements. After forming the two paths that start with literal 'A' the algorithm returns to the cell with row '0' and column '1', keeping in the list just the index '0'. Of course the algorithm travels this row finds the element 'B' in row '0' and column '3'. So, all procedure explained above is applied again as shown the Fig. 2.d and Fig. 2.e. The process ends when it returns to the initial row '0' and all cells this row were visited.

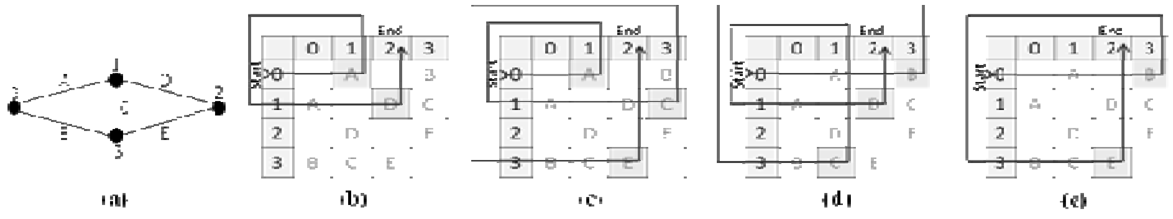

Fig. 2 – Matrices for explaining the algorithm able to generate all paths of a graph.

The next step consists in comparing each path with the products that compose the original expression. If a path that does not belong to SOP has been introduced, a routine checks if this path is sensitized or not sensitized. When thinking in a transistor network, a path cannot be sensitized if it contains both polarities, for example 'A' and '!A'. In order words, this path is not a valid path. If the new path introduced is not sensitized he is accepted, because it does not change the logical behavior of the circuit. Otherwise, the graph needs to be restored to step before the optimization that generated the new path, and this optimization is discarded. For this, a restore routine is invoked, this is responsible for recovering the edges and vertices that were eliminated from the graph and reconnect them.

To perform the recovery process it is necessary to store some information as the literal represented by removed edges and what vertices were merged. Finally, if necessary, with this information the graph can be recovered without compromising the functionality of the circuit which it represents. Notice that all original products of the Expression (1) are present in the graph of the Fig. 1.h. However, by sharing edges, some new paths were also introduced. All these paths are allowed because are paths that cannot be sensitized. Another interesting fact is that the proposed approach may derive Wheatstone bridge networks like methods proposed by [1] and [2]. The example illustrated in Fig. 1.h presents some bridge configuration. It is a benefit over optimization approaches based on factorization that can only derive series-parallel networks.

4. Experimental Results

Our algorithms were implemented in Java language. As output, the technique shows the optimized networks using the Prefuse graphics library [7] and generates a Spice netlist of the optimized circuit. To describe our edges sharing method we used the Exp. (1), referring to a 'XOR' with four inputs. The achieved network was compared to the result obtained by others techniques described in [5], as BDD, OpBDD, LBBDD, CSP and to the good-factor algorithm from SIS Software [8]. Our method reaches the same result like the BDD, OpBDD and LBBDD methods, with 12 transistors, overcoming the SIS with 16 transistors and CSP, NCSP with 22 transistors.

Finally, the set of 4 input p-class logic functions was used as benchmark to evaluate our proposed algorithm. This set is composed by 3982 logic functions. Each logic function was applied to SIS software as well as to our solution. When running in SIS, the two available algorithms were used, the quick-factor and the good-factor. However, our proposed method was able to deliver better solutions, reducing the total number of switches in the networks as Tab. 1 illustrates. This is due to the ability of generating networks with Wheatstone bridge arrangements. Our approach was capable to reduce up to 4 transistors in some generated networks if compared to the SIS. On the other hand, the good-factor achieves some smaller transistor networks. On these 130 cases our algorithm was not able to generate bridge configuration, generating networks that are purely series-parallel arrangements.

Tab. 1 – Results for the set of 4 input p-class logic functions.

	Total transistor count
Our solution	35598
Good-factory	37723
Quick-factory	38341
Our solution compared to SIS	**# of logic functions**
decreasing transistor count	1644
exact same transistor count	2208
increasing transistor count	130

5. Conclusions and Future Works

This paper presented an edges sharing method to derive optimized transistor networks. The algorithm was implemented in Java language and a graph interface using Prefuse library is available. To describe our algorithm step by step, we use an 'XOR' with 4 inputs. The optimized network presents the same result of the methods in [5] for a ‘XOR’ with 4 inputs, surpassing the SIS solution. Nevertheless, when using the set of 4 input p-class logic functions, our solution is able to perform a considerable reduction of the total transistor count. Moreover, it is capable to deliver 1644 networks with less transistor count, if comparing to the good-factor solution, reduce up to 4 transistors in some networks. The optimized transistor networks generate by our approach are validated ensuring the logical behavioral of the network. As future work we intend to evaluate the complexity of the algorithm. Also, we intend to compare the proposed solution with the method described in [4].

6. References

[1] J. Zhu et al. On the Optimization of MOS Circuits. IEEE Transactions on Circuits and Systems: Fundamental Theory and Applications. (1993), 412-422.

[2] D. Kagaris et al. A Methodology for Transistor-Efficient Supergate Design. IEEE Transactions On Very Large Scale Integration (VLSI) Systems. (2007), 488-492.

[3] Brayton, R. K. Factoring logic functions. IBM J. Res. Dev. 31, 2 (1987), 187-198.

[4] Mintz, A. and Golumbic, M. C. Factoring boolean functions using graph partitioning. Discrete Appl. Math. 149, 1-3 (2005), 131-153.

[5] Da Rosa Jr, L. S. Automatic Generation and Evaluation of Transistor Networks in Different Logic Styles. PhD Thesis PGMicro/UFRGS, Porto Alegre, Brazil. (2008), 147 p.

[6] Da Rosa Jr., L. S.; Marques, F. S.; Schneider, F.; Ribas, R.P.; Reis, A. I. A Comparative Study of CMOS Gates with Minimum Transistor Stacks. Symposium on Integrated Circuits and Systems Design. (2007), 93–98

[7] Prefuse.org. The Prefuse Visualization Toolkit. [Online] Avaliable: http://prefuse.org/ [Acessed: Mar. 25, 2010].

[8] Sentovich, E.; Singh, K., Lavagno; L., Moon; C., Murgai, R.; Saldanha, A., Savoj; H., Stephan, P.; Brayton, R.; and Sangiovanni-Vincentelli, A. SIS: A system for sequential circuit synthesis. Tech. Rep. UCB/ERL M92/41. UC Berkeley, Berkeley. (1992).