TRANSISTOR NETWORKS DESIGN USING A GRAPH-BASED APPROACH

Vinícius N. Possani, Éric F. Timm, Luciano V. Agostini, Leomar S. da Rosa Jr.
{vpossani.ifm, erict.ifm, agostini, leomarjr}@ufpel.edu.br

Grupo PET Computação UFPel
Grupo de Arquiteturas e Circuitos Integrados – GACI
Universidade Federal de Pelotas
Pelotas – Brasil

ABSTRACT

The number of transistors required for implementing a logic function is an essential consideration in digital VLSI design. While the generation of a series-parallel network can be straightforward once a minimized Boolean expression is available, this may not be an optimum solution. This paper proposes a graph-based solution for minimizing the number of transistors that compose a network. The algorithm starts from a sum-of-products expression and can achieve non-series-parallel arrangements. Experimental results demonstrate the efficiency of the approach when compared to the quick-factor algorithm implemented in the SIS software.

1. INTRODUCTION

Nowadays, VLSI design has definitely established a dominant role in the electronics industry. Automated tools have held designers to manipulate more transistors on a design project and shorten the design cycle. In particular, logic synthesis tools have contributed considerably to reduce the cycle time. In full-custom designs, manual generation of transistor netlists for each functional block is performed, but this is an extremely time-consuming task. In this sense, it becomes comfortable to have efficient algorithms to derive transistor networks automatically.

The common technique to optimize a transistor network is based on factorization [1-2]. In this procedure 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. Alternative methods are also available in the literature. 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 [3] or to compose a new graph with a reduced number of edges [4].

In this sense, this paper investigates a graph-based method to generate transistor networks. In our approach, the input Boolean expression is translated into a graph that is later optimized through edges sharing.

The remainder of this paper is organized as follows. Section 2 presents the proposed method to optimize transistor networks. Section 3 describes the implemented tool and presents the experimental results. The conclusions are presented in Section 4.

2. GRAPH-BASED APPROACH

The proposed graph-based approach accepts a sum-of-products (SOP) expression as input. In order to translate the expression to a graph, a parser is needed. The basic idea of this parser consists, initially, in separating all products that compose the SOP. In the sequence, each literal present in the product is extracted and stored in a vector. For each product a vector is created. At the end, the parser will deliver '*n*' vectors for the '*n*' products that compose the input Boolean expression. Figure 1 illustrates all vectors obtained from the Expression 1 that represents an input SOP.

!A*C*E*F*!G*H + !A*B*F*!H + A*!B*C*!G*H (Exp.1)

!A	C	E	F	!G	H
!A	B	F	!H		
A	!B	C	!G	H	

Fig. 1 – Vectors representing the products from Expression 1.

Once the list of vectors is obtained, they are organized according to the number of literals that compose them. Figure 2 exemplifies this situation.

Afterward, it is started the assembly of the graph by removing a vector at a time from this list and creating an edge in the graph for each literal found in the vector. Notice that this operation will create a set of edges connected in series. This is demonstrated in Figure 3, where it is also possible to see the vertices that make connections between the pairs of edges.

!A	C	E	F	!G	H
A	!B	C	!G	H	
!A	B	F	!H		

Fig. 2 – Ordered and organized vectors representing the products from Expression 1.

Fig. 3 – First product as a graph.

In the sequence, another vector is loaded from the list of vectors and placed in the graph. Figure 4 illustrates that for the second vector in the list. All paths in the graph are traversed in order to recognize identical vertices (vertices that represent same literals). If this condition is verified in the graph, then the identical edges are carried to the beginning of the graph and they are shared. This operation leads to a decrease of edges count. This is exemplified in Figure 5, where the edges ‘C’, ‘!G’ and ‘H’ are merged.

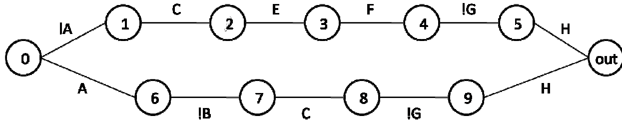

Fig. 4 – Graph composed by the two first products.

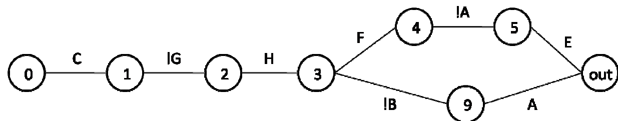

Fig. 5 – Optimized graph for the two first products.

This procedure is performed until the list of vectors is completely empty.

Figure 6 shows the last vector added to the graph. Although there are edges candidates to be merged, ‘!A’ and ‘F’, in this case it is not possible to perform the optimization. It is duo to the fact that vertex ‘3’ is a separation point between the two paths.

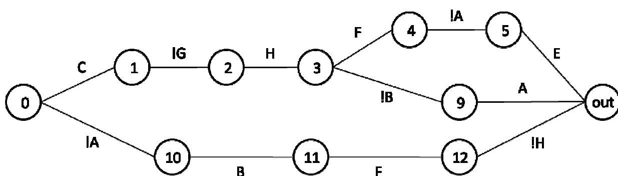

Fig. 6 – Graph with the last product added.

After finishing the optimization process, it is started again, but now the graph is traversed from the end to the beginning in order to optimize the literals that could not be moved and merged before. For this, the vertices ‘0’ and ‘out’ are changed of place, as if the graph has been turned 180 degrees.

Now, the literal ‘!A’ and ‘F’ can be brought to the beginning of the graph in order to be merged, each one with its equivalent literal. Figure 7 illustrates this procedure.

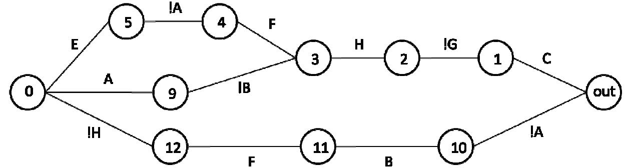

Fig. 7 – The graph has been turned 180 degrees.

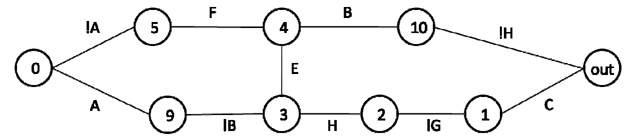

Fig. 8 – Obtained graph after the final optimization.

To guarantee that sneak-paths (forbidden paths) are not introduced in the graph, a routine that traverses the graph and compares it to the original products of the expression is regularly invoked. This is necessary because if a sneak-path is introduced, the graph will not be a true representation of the input Boolean expression.

Notice that all original products of the SOP are present in the graph through the paths ‘!A F B !H’, ‘!A F E C !G H’ and ‘A !B C !G H’. However, by sharing edges a new path ‘A !B E B !H’ was also introduced. This path is allowed since it not modifies the logical behavior. When thinking in a transistor network, this new path cannot be sensitized because it contains transistors controlled by variable ‘B’ in both polarities. In order words, this is not a valid path.

The validation routine uses Latin arrays to generate all the existing paths in the graph. These paths are compared to a list that contains all paths of the graph before the optimization process. If a new path is inserted, then it is verified if it contains the direct literal and the complemented one. If the new path does not have them, it means that it is an invalid path. In that case this optimization is refused and the method continues to run.

An interesting detail is that the proposed approach may derive bridge networks like methods proposed by [3] and [4]. The example illustrated in Figure 8 presents a bridge configuration (through edge ‘E’). It is a benefit over optimization approaches based on factorization that can only derive series-parallel networks. For several Boolean functions the series-parallel arrangement is not the most optimized solution [5].

Tab. 1 – Results for 10 randomly choose Boolean functions with 7 input variables.

Function	# literals SOP form	# literals SIS	# edges Soptimizer	% of gain
F1	133	80	59	26,25
F2	92	70	44	37.44
F3	78	62	39	37.10
F4	150	78	70	10.26
F5	119	82	55	32.93
F6	71	44	38	13,64
F7	170	76	82	-7,89
F8	135	78	62	20.51
F9	111	74	59	18.75
F10	97	64	52	18,75

3. EXPERIMENTAL RESULTS

The proposed method has been implemented in Java language using the NetBeans IDE 6.5.1. A tool containing the core algorithms and the graphics interface was developed. The graphics interface uses the Prefuse library [6]. Figure 9 illustrates it for the example from Expression 1. Also, the tool presents a Spice netlist output module that is capable to print Spice files to be used in electrical simulators.

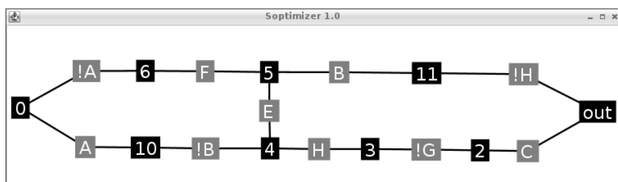

Fig. 9 – Graphical interface of the developed tool.

In order to evaluate the proposed approach, 10 Boolean functions with 7 input variables were randomly chosen. They were introduced in SIS software [7] and extracted in their SOP form. Table 1 presents the obtained results. The total numbers of literals for the SOP form are described in column "# literals SOP form".

The expressions were factorized using the quick-factor algorithm from SIS. The results are shown in column "# literals SIS". Results obtained using the proposed approach are shown in column "# edges Soptimizer". The obtained gain over the SIS software are shown in column "% of gain".

In most cases the proposed approach achieved smaller results. Analyzing the obtained networks we identified several bridge configurations in the arrangements delivered by the tool. On the other hand, SIS software delivers optimized expressions composed by 'AND' and 'OR' operators. In this case only series-parallel networks can be implemented, representing an overhead in terms of area (transistor count).

For function F7, the proposed algorithm was not able to deliver a smaller solution. The point in that case is the ability of the proposed algorithm to achieve bridge configuration. For this input expression the algorithm could not find bridges. We believe that it is related to the SOP ordering that is used to compose and to optimize the graph. As mentioned before, we start using products with a larger number of literals. However, when the products that compose the SOP present almost same number of literals, we choose them randomly. This situation could be leading for this kind of result.

4. CONCLUSION AND FUTURE WORKS

This paper presented a graph-based approach to generate optimized transistor networks. The proposed solution is able to deliver bridge networks. The algorithm was implemented in Java language and a graph interface using Prefuse library is available. 10 Boolean expressions with 7 input variables were randomly chosen to be used as benchmark. The results demonstrated that the proposed approach can delivery networks with a transistor reduction up to 37.44% if compared to the quick-factor algorithm from the SIS software.

As future works we intend to investigate the impact of selecting different products ordering to start the optimization process. Also, we intend to compare the proposed solution with the method described in [4].

5. REFERENCES

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

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

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

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

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

[7] 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).